UNIVERSITÉ PARIS XI

U.E.R. MATHÉMATIQUE

91405 ORSAY FRANCE

100-7420

APLASM 73

SYMPOSIUM D'ORSAY SUR LA MANIPULATION DES SYMBOLES ET L'UTILISATION D'APL

VOLUME DEUX

LE PROJET LIMA

EDITÉ PAR P. BRAFFORT ET P. MERISSERT-COFFINIERES

CONTENTS

VOLUME ONE : THE AUTOMATH PROJECT

VOLUME TWO : THE LIMA PROJECT

VOLUME THREE : RESEARCH ON APL

LE SYMPOSIUM APLASM 73

Les 20 et 21 décembre I973, le Département de Mathématiques de l'Université Paris-Sud (Laboratoire Al Khowarizmi) organisait à Orsay un Symposium International consacré aux problèmes de la manipulation des symboles en mathématique pure et à l'utilisation du système APL.

Plus de cinquante participants venus de huit pays différents furent acceuillis par G. POITOU et participèrent aux sessions présidées par M. DEMAZURE, E.HAEGI, G. MARTIN, J. DELBREIL.

Une introduction générale au projet LIMA et aux problèmes généraux abordés au cours du Symposium est publiée séparement (note ECSTASM N°1).

Nous publions, avec le concours de l'IRIA (*) les communications présentées pendant ces deux journées en les regroupant en trois volumes qui correspondent aux trois pôles d'intérêts principaux.

Certaines communications n'étaient pas disponibles pour publications,par contre nous avons ajouté plusieurs textes correspondant à des travaux effectués postérieurement au Symposium et qui permettent de parfaire l'homogénéité de l'ensemble.

P.B , M.D.

(*) Contrat SESORI 73 021

II.0 INTRODUCTION AU DEUXIEME FASCICULE

par

P. BRAFFORT

On trouvera dans ce fascicule, un certain nombre d'exemples de ce qu'on peut grouper sous le terme général de "système LIMA".

Il s'agit donc de systèmes de manipulation de symboles spécialisés ce qui les distingue, notamment, des systèmes AUTOMATH.

L'avantage est évidemment que l'on peut choisir, pour les objets formels que l'on manipule, une représentation interne soigneusement étudiée pour améliorer les performances des "manipulateurs".

L'inconvénient est, bien entendu, le manque de souplesse qui rend très difficile toute extension du système.

Nous présentons ici deux applications caractéristiques des systèmes LIMA :

LIMA A, système de manipulation algébriques

LIMA O, système de manipulation des limites.

ainsi que deux applications plus orientées vers les problèmes linguistiques de l'informatique :

- le système de vérification de la correction des programmes développé à l'Université de Entschede.
- le système d'analyse syntaxique et sémantique développé à l'Université de Florence.

Tous les systèmes présentés dans ce chapitre souffrent évidemment des limitations liées à la nature même d'APL , qui est leur commun langage d'implémentation.

Il n'y a pas trop lieu de s'en préoccuper tant qu'on demeure au niveau de l'expérimentation.

Mais dans la mesure ou l'on se propose de construire des systèmes destinés à une large utilisation, il convient de revoir de très près le langage de programmation lui-même tant en ce qui concerne la structure des données qu'en ce qui concerne les techniques de compilation et d'interprétation.

Ces points sont abordés dans le troisième fascicule.

LIMA - A

Denis FELDMANN*

* CNRS
Laboratoire AL KHOWARIZMI

Ce travail a été réalisé dans le cadre du contrat SESORI 73-02I

I. Introduction.

Dans le cadre du projet LIMA général (construction d'un langage intéractif pour les mathématiques algorithmiques), la phase 0 est consacrée à la réalisation de LIMA particuliers, destinés à déterminer avec précision les besoins variés des utilisateurs, et à vérifier sur des cas particuliers l'adéquation des options principales du projet LIMA (choix du langage APL et de sa syntaxe, symbolique très développée, etc...) aux problèmes courants du mathématicien [1] .

LIMA A est ainsi un langage destiné à manipuler des "expressions algèbriques", qui a été conçu dans le but de pouvoir aisément exprimer et résoudre certains problèmes de manipulation de polynômes, dont le prototype donné par un mathématicien du groupe ECSTASM, était le suivant : on considère, dans Z^8 , les 256 vecteurs n'ayant que des 0 ou des 1 pour coordonnées, que l'on note x_i ($1 \leq i \leq 256$) (dans un certain ordre). On veut, sur l'ensemble des polynômes à 256 indéterminées sur Z , considérer les 256 relations d'équivalence $(R_i) : X_i^2 \cong \sum_{j<i} \langle x_i , x_j \rangle X_j$ (où $\langle a,b \rangle$ est le produit scalaire dans Z^8) , et le problème consiste à trouver les polynômes réduits équivalents à des polynômes tels que $X_2^3 X_{34}^5$ par exemple.

(L'origine de ce problème, une conjecture très abstraite en géométrie algébrique, est un bon exemple des demandes inattendues de calculs simples mais très pénibles, que peut être amené à formuler le mathématicien pur).

A partir de ce problème précis, une version "initiale" de LIMA a été réalisée, puis élargie en fonction des premières critiques des utilisateurs. Toutefois, la version qui va être décrite est encore loin de satisfaire tous les besoins du mathématicien en matière de manipulations algé-

briques (ni même la plupart d'entre eux) la section algèbrique du système LIMA définitif, même si elle conserve la plupart des idées de LIMA A, devra par exemple y adjoindre un système de résolution d'équations (à coefficients littéraux).

Les problèmes principaux résolus par LIMA A sont essentiellement linguistiques : comment communiquer sous une forme compacte la liste des relations d'équivalence donnée plus haut, par exemple ?

Il est formé, d'une part, de programmes de calcul dans un anneau de polynômes arbitraire (c'est-à-dire que l'anneau des constantes, noté A, peut être quelconque, sous réserve que les calculs y soient déjà soumis à des algorithmes), ayant un nombre quelconque d'indéterminées. Cet anneau sera noté B dans la suite. D'autre part, un analyseur assez complexe manipule un langage qui permet, comme on le verra au §4, d'exprimer aisément les "longues" expressions citées plus haut.

II. L'algèbre $A[X_1, X_2, \ldots, X_n]$.

Structure

L'algèbre des polynômes à n indéterminées sur A est définie formellement comme l'ensemble des suites d'éléments de A indéxées par $\mathbb{N}^n$ dont tous les éléments sauf un nombre fini sont nuls. On la munit des lois d'addition comme module sur A $((a_i) + (b_i) = (a_i + b_i))$ et de la multiplication définie par la formule :

$$((a_i) \times (b_j))_{(i_1,i_2,\ldots,i_n)} = \Sigma\, a_i \times b_j \quad \text{pour } (i) + (j) = (i_1,i_2,..,i$$

Pour ces deux lois, on voit aisément que l'on a bien une structure d'algèbre sur A ; elle satisfait, si la multiplication de A est commutative, à une X "propriété universelle" qu'on peut décrire en introduisant les indéterminées X_i $(1 \leqslant i \leqslant n)$: ce sont les suites nulles pour tous les indices sauf $(0,0,\ldots,0,1,0,\ldots)$ où le 1 se trouve à la i ème place.

Alors, si on donne des valeurs indéterminées, elles se prolongent d'une manière et d'une seule à des valeurs des pôlynômes ([2]).

Si A est un anneau commutatif unitaire, l'addition, la multiplicatior et les substitutions se comportent donc de façon régulière, Sinon, les formules données plus haut conservent un sens (si l'addition est associative et commutative) ; mais la structure résultante n'a que peu d'intérêt ; on s'est efforcé dans LIMA A de donner aux programmes effectuant ces opérations la plus grande généralité possible, mais cependant l'utilisateur qui voudrait travailler avec une multiplication dans A non distributive sur l'addition par exemple aurait intérêt à vérifier les détails du programme PMULT !

Exponentiation.

LIMA A n'utilise que très peu de résultats théoriques, mais le calcul des exponentielles de polynômes a été simplifié par les remarques suivantes : Si $P = T_1 + T_2 + \ldots + T_k$, P^m est la somme de tous les termes de la forme :

$$T_1^{n_1} \times T_2^{n_2} \times \ldots \times T_k^{n_k} \quad \text{avec} \quad n_1 + n_2 + \ldots + n_k = m$$

multipliés par les coefficients multinominaux $\dfrac{m!}{n_1!\, n_2! \ldots n_k!}$

Cette formule permet déjà une simplification importante, mais si A est intègre et si m est la caractéristique de A, c'est-à-dire le plus petit a pour lequel $a \times y = 0$ pour tout y de A , on a $P^m = T_1^m + T_2^m + \ldots + T_k^m$ (Ce qu'on démontre en remarquant que m (qui est alors premier) divise tous les coefficients multinomiaux saut ceux égaux à 1).

Ce cas est suffisamment important pour être testé séparément, et l'utilisateur a donc intérêt à fournir la caractéristique de A quand il la connait.

Représentation canonique.

On sait que tout élément de $A[X_1 , X_2 , \ldots , X_n]$ s'exprime comme somme de prdduits d'indéterminées et d'éléments de A , termes appelés monômes. Le programme travaille naturellement dans cette base, où les algorithme de calcul vont être donnés au § suivant.

Dérivation.

Une notion de dérivée formelle existe pour les anneaux de polynômes, en l'absence même de toute structure topologique sur A . L'une de ses

définitions possibles consiste à plonger B dans B[Y] , et à définir la dérivée du polynôme P par rapport à l'indeterminée X comme le coefficient de Y (dans B[Y])du polynôme $P(;; X_i + Y ;...) - P(;; X_i ;...$

LIMA A pourrait appliquer directement cette définition, mais il est plus commode d'utiliser la linéarité de la dérivation, et le fait que $\frac{d(X^n)}{dX} = nX^{n-1}$ (En n'oubliant pas d'annuler les termes pour lesquels n est multiple de la caractéristique de l'anneau A).

Plus généralement, une théorie de la différentiation est possible, et certains utilisateurs de LIMA A auraient aimé pouvoir manipuler la différentielle totale DP , définie par :

$$DP = \sum_i (P'_{X_i}) \, DX_i$$

où les DX_i sont de nouvelles indéterminées (en fait, cela veut dire que l'on commence par plonger B dans un nouvel anneau de polynômes C , ayant deux fois plus de variables que le précédent). Mais outre que cela eût posé de difficiles problèmes de notation, et que l'adjonction de nouvelles variables, sans être impossible, est assez pénible en LIMA A , la répétition de ce processus pour atteindre les différentielles d'ordre supérieur paraissait presque incompatible avec les options déjà prises. On y a donc provisoirement renonné, se réservant d'y revenir lorsqu'un système plus élaboré permettra de meilleures manipulations (internes) des objets et des noms.

III. Techniques de calcul.

Représentation interne.

Un polynôme est représenté par une matrice T et un tenseur H

H est la "pile des coefficients" : on suppose que tous les coefficients de A sont représentés en APL par des tenseurs de mêmes dimensions (fixées par l'utilisateur en début de session) et $\overset{*}{H}$ a une direction de plus, dans laquelle les coefficients du polyhôme s'empilent. T est formée des codes des monômes, qui sont des vecteurs, et à la même longueur d'empilement que H . Les termes nuls n'apparaissent pas, et le polynôme nul est codé par une matrice et un tenseur de l'ongueur 0 .

Le code des monômes est une option essentielle de LIMA A . Si le nombre des indéterminées n'est pas trop grand, le codage se fait en modèle : chaque monôme est codé par le vecteur des exposants de toutes les indéterminées. Sinon, on passera en mode 2 , où le code fait précéder chaque exposant de l'indice de l'indéterminée correspondante.

D'autre part, si les entiers (indices et exposants) de ce codage ne sont pas trop grands,on peut obtenir un codage supplémentaire (et un énorme gain de place en mémoire) en passant en mode 3 et 4 respectivement : les vecteurs sont alors considérés comme des entiers écrits en base assez grande, chaque monôme est représenté par l'entier obtenu.

Dans ces modes, l'utilisateur doit être certain qu'exposants et indices ne dépasseront pas la limite qu'il fixe en passant en mode 3 ou 4 , car sinon les erreurs commises ne sont pas toutes diagnostiquées par le système.

Dans tous les cas, les algorithmes que l'on va décrire nécessitent que les termes des polynômes soient classés. On utilise donc un ordre lexico-

graphique, (qui se ramène en modes 3 et 4 à la simple comparaison de deux entiers), et qui sous entend que les variables sont classées par ordre d'indices. Ainsi,

$$X_1 > (X_2)^3 X_4 > (X_2)^3 (X_6)^2 > (X_3)^4 \quad .$$

L'algorithme de tri.

Une fois effectuée une opération sur des polynômes, le résultat n'est en général plus classé ; or pour regrouper les termes identiques, il faut à peu près autant d'opérations que pour tout reclassér c'est pourquoi on s'est arrété à cette deuxième solution, beaucoup plus aisé dans les manipulations qu'elle permet. Mais pour classer un ensemble de n données, il faut à première vue (en employant la méthode évidente consistant à rajouter un terme à chaque fois) $n^2/4$ opérations ; la technique sophistiquée que nous allons décrire, n'en prend que $k\, n \log n$ (où k est une constante) ; toutefois, il est plus rapide d'employer la méthode naïve X si n est très petit.

Tout d'abord, on remarque que la réunion de deux listes déjà classées se classe aisément en comparant les deux plus petits termes : on met le plus petit aomme premier élément de la liste réunion, et on recommence avec ce qui reste. Ainsi, chaque élément n'est testé qu'une fois.

On procède alors dans le cas général par dichotomie : on commence par classer les éléments deux par deux, puis on réunit les listes deux par deux (obtenant des ensembles classés de 4 éléments) et on recommence, pour avoir des ensembles classés de 8, 16,... éléments.

Au cours de cet algorithme, on peut même regrouper les terme semblables qui apparaissent, à condition de garder trace des débuts et fins des sous-listes déjà classés.

Addition

L'addition n'est qu'une réunion de deux polynômes, il suffit donc d'effectuer les additions dans A au fur et à mesure qu'elles se présentent (et d'éliminer les termes dont le coefficient devient nul).

Multiplication

Elle se fait en deux phases : d'abord, on calcule l'ensemble des produits des termes deux à deux, puis on applique l'algorithme de tri (légèrement modifié, car des suites partielles déjà classées assez longues résultent de la multiplication d'un terme du premier polynôme par le second).

Exponentiation

En dehors du cas trivial où P n'a qu'un terme, et du cas où l'éxposant est la caractéristique de A (ou un de ses multiples), on applique directement la formule multinomiale, qui de plus donne un résultat déjà classé si P n'a que deux termes. Sinon, on termine par l'algorithme de tri.

Substitution

Il ne s'agit que des substitutions de la forme : P remplace $(X_i^{n_1} . X_{i_2}^{n_2} \ldots X_i^{n_k})$ dans Q . La recherche du monôme $(X_1^{n_1}, X_2^{n_2} \ldots X_i^{n_2})$ dans les termes de Q (pas si facile en mode 2 et 4) une fois faite, il faut calculer P^n où n est le nombre de fois où le monôme apparait dans un terme ; et substituer; le calcul de P^n n'est fait qu'une fois pour les petits exposants $(n \leq 4)$ et le résultat est stocké ; une fois les substitutions faites, on trie le résultat.

Dérivation

On dérive " terme à terme" la recherche de l'exposant de X_i est très aisée, et le polynôme reste classé après la dérivation.

IV Le langage de LIMA A.

Les trois niveaux du langage.

LIMA A est essentiellement un langage conversationnel. Le mode fondamental est donc, comme en APL, le mode exécution, où chaque instruction est immédiatement analysée, et le résultat imprimé. Mais d'autres commandes réclament une exécution différée : ce sont d'une part les sous-programmes, pour lesquels les conventions sont analogues à celle d'APL, de l'autre les règles de "simplifications", qui ont une syntaxe spéciale et qui ne sont analysées qu'au fur et à mesure des besoins. Enfin, l'ensemble des commandes-systèmes est plus développé que dans les LIMA précédents, et possède un aspect intéractif qui n'existe pas en APL.

Le mode exécution.

a) Alphabet.

Les caractères du mode exécution sont : les lettres et les chiffres ; les symboles opératoires + , - , × , * , !) les opérateurs vectoriels ∼, o et, ; les "opérateurs " ',/, ⍀ et ⍴ ; les relations = , ≠ ; les séparateurs (,) ; [,] ; < , > ; Λ ; ∈ le ; symbole fonctionnel ⍎ , l'affectation ← et le "quad" ⎕ , la "substitution" → .

b) Vocabulaire

On verra, dans le paragraphe consacré aux commandes-système, comment sont affectées les lettres respectivement aux constantes de l'anneau A aux indéterminées des polynômes, et aux polynômes eux-mêmes. Dans tous les cas, un nom se compose d'une lettre non soulignée, où de l'une des trois lettres P,Q ou R , suivie éventuellement soit

d'un entier explicité (sans blanc séparateur) (par exemple A ; X2 ; Q457) ; soit d'une expression entre crochets (Par exemple A[1] ; B[N + 2] ; X[3 ; M ; 8]) .

Les seuls noms de fonctions permis sont de laforme Φ suivi d'un entier explicité, ou d'une expression entre crochets. (ie Φ3 ; Φ[I + 4]) .

Le système ne distingue pas entre polynômes et entiers ou vecteurs d'entiers au niveau des noms ; les deux types sont obtenus par des affectations et des messages d'erreur ne sont produits que si les opérations demandées n'ont pas de sens pour le type d'un des arguments.

c) Syntaxe

Comme le veut le projet LIMA dans son ensemble, la syntaxe adoptée est la syntaxe APL : pas de priorité entre les opérateurs, analyse allant de la droite vers la gauche. Mais la relative complexité de LIMA A a amené à ajouter d'autres conventions. Tout d'abord, priorit est donnée aux séparateurs < , > : ils enferment une constante de l' anneau A , écrite dans un langage séparé du reste de LIMA A , que l'utilisateur doit éventuellement (dans un des modes USER) permettre de déchiffrer.

Ainsi, dans le mode COMPLEX, une telle expression est de la forme <(A) ± (B)ι> (A et B pouvant être nuls et donc omis) ; l'option COMPLEX étant fournie avec LIMA A , un petit analyseur se charge de ces expressions.

Entre les séparateurs < et > , n'importe quel caractère APL (même ceux ne faisant pas partie de l'alphabet de LIMA A) est admis, à l'exception des séparateurs (,) , [,] , < et > . (Ce, pour ne pas compliquer à l'excès la tâche de l'analyseur, et ne pas risquer d'ambigüités).

De même, les crochets d'indexation (c'est-à-dire ceux précédés par une lettre indexable (constante de l'anneau A ou indéterminée) ou par Φ) sont évalués prioritairement, et l'index (ou les indexes) obtenus, le numéro de la variable est calculé à l'aide des indications fournies par les commandes-système.

Les crochets suivant un nom de polynôme annoncent une substitution : si les trois indéterminées de l'algèbre B sont X,Y et Z (dans cet ordre) la notation P [3 ; (Y 4) + 1,Z] désigne le résultat de la substitution de 3 à X , puis de $Y^4 + 1$ à Y dans P (une substitution simultanée serait possible, mais necéssiterait une complication du programme de substitution qui n'a pas été jugée indispensable ; les inconvénients résultant de ce fait sont palliés dans la mesure du possible par la possibilité de permuter les variables à l'aide de l'opérateur ⍉) Si le nombre d'indéterminées de B dépasse de beaucoup celui des variables de P (comme c'est souvent le cas dans l'exemple de l'introduction), cette notation risque de devenir malcommode, on est alors autorisé à ne faire figurer qu'un nombre de substitutions égal à celui des variables de P (Ce qui nécessite de connaitre celles-ci)

Pour des substitutions plus complexes, la notation [M → Q] /P où M doit être un monôme, est employée ; la justification de ce type

de substitution (C'est-à-dire les problèmes posés par le passage à un anneau quotient de l'algèbre B) sera donnée dans le § consacré aux règles de "simplification".

A partir de ces deux exemples, une réflection théorique sur le rôle des crochets en LIMA A (et en APL) s'imposait ; de même que les parenthèses enveloppent des expressions ordinaires du langage, qui doivent simplement être évaluées prioritairement dans l'analyse, et que les séparateurs < et > enferment des expressions analogues, mais écrites dans un code spécial, les crochets enferment des fragments, c'est-à-dire des suites sans signification par elles-mêmes (dans le LIMA considéré); mais susceptibles de jouer le rôle de l'opérateurs pour les objets du langage. (De façon formalisée, on peut dire que les objets (linguistiques) d'un LIMA forment une algèbre, dont les opérations internes sont celles du domaine mathématique considéré, ainsi que certaines opérations "linguistiques" (affectation, modification de structures internes,...), et sur lesquels les fragments entre crochets opèrent extérieurement). Ainsi, en APL , les crochets enferment des listes de tableaux, structures "interdites" en APL proprement dit et dont l'introduction (par exemple dans le système proposé par Ghandour et Mezel [3]) rend l'indexation d'APL inutile (ils lui substituent X l'opérateur "slice"). Les fragments considérés ainsi par LIMA A sont plus hétérogènes, ceux que nous venons de voir sont des listes d'ordres. Il serait sans doute possible de les remplacer par des listes de polynômes, mais il resterait encore à définir un opérateur ternaire de substitution, qui masquerait d'ailleurs mal le fait que les monômes exigés plus haut ne sont pas en réalité tant des cas particuliers de polynômes que des listes de noms d'indeterminées.

En tout cas, une fois cette interprétation admise, rien ne s'opposait plus à d'autres utilisations analogues des crochets ; LIMA A en connaît une troisième, c'est la liste des noms de variables. Ainsi, l'expression [X ; Y ; Y ; Z]'P désigne le polynôme dérivée quatrième : $P''''_{x,y,y,z}$ et l'opérateur de transposition ⍉ a été utilisé pour effectuer les permutations de variables : [Z ; Y ; Y ; X] ⍉ P , avec P = X^2 + Y + 2Z - T est : -X + 3Y + Z^2

La multiplication a été élidée ; (c'est-à-dire que l'oubli de X ne provoque pas de confusions) ; il s'agissait là d'une expérience de manipulation de l'analyseur synatxique, qui ne saurait de toute façon présenter beaucoup d'intérêt dans le cadre d'un LIMA général (en fait, elle n'en a même pas en APL), mais qui est néanmoins agréable pour alléger l'écriture d'un langage où la multiplication est privilégiée (théorie des groupes, manipulations algèbriques) et où les noms d'une lettre sont la majorité. On pourrait conserver une telle option sous contrôle d'utilisateurs avertis.

Nous avons volontairement passé sous silence l'opérateur ρ ; sa syntaxe et sa sémantique très particulières se verront réserver un paragraphe séparé.

d) Sémantique

On n'a pas juger utile de séparer l'addition des constantes de celles des polynômes, par exemple ; en effet, tous les domaines d'opérateurs considérés (polynômes, entiers, constantes de l' anneau A) sont plongés "canoniquement" les uns dans les autres (si du moins l'anneau A est unitaire) et seuls les vecteurs d'entiers sont provisoirement exclus de la plupart des opérations, en attendant qu'un LIMA A plus général

manipule les tableaux de polynômes. Une première version de LIMA A connaissait la "multiplication extérieure" des polynômes par les constantes de A (en effet celle-ci correspond à un programme spécial, car la représentation interne facilite beaucoup de cas particulier) ; mais il est clair en fait qu' il est plus rentable de faire tester ce cas par la machine que de demander à l'utilisateur de le spécifier. Ainsi, seuls l'opérateur * (exponentiation) doit avoir son argument de droite entier ; et l'opérateur ! , calculant les coefficients binomiaux, est réservé aux entiers. ~ et, ont leurs significations vectorielles usuelles (en APL) et 0 a la signification de ⌽ en APL (prise en LIMA A pour les noms de fonctions

Le "quad" et l'affectation ont la même syntaxe et la même sémantique qu'en APL .

On a vu la syntaxe (et la sémantique) spéciale des opérateurs /, ' et précédés de crochets au § précédent, mais ils peuvent avoir aussi un argument gauche ordinaire (qui doit alors être un vecteur d'entiers) ; dans ce cas, pour les opérateurs ' et ⍀ , ce vecteur est formé des numéros des variables (l'ordre de celles-ci étant connu de l'utilisateur, ce n'est pas un inconvénient trop grand) ; pour l'opérateur / , ce vecteur est celui des numéros des règles de substitutions, lesquelles sont alors appliquées (dans cet ordre) à X l'argument de droite.

Concernant ce dernier point, on peut se demander si une syntaxe plus "fonctionnelle", par exemple VRP , où R serait le nom d'un "programme" de règles, n'aurait pas été plus simple. En réalité, on verra dans les applications que le numéro d'une règle y joue le rôle d'un argument scalaire, et que (comme en APL) il est souvent possible de l'étendre à un argument vectoriel, mais pas toujours ; on peut penser que la résolution de ce dernier problème n'est pas liée à des simples questions de syntaxe ou de structure des objets du langage, mais plutôt à la notion de processus

parallèle (puisqu'un argument vectoriel revient à la demande de n exécutions parallèles pour n arguments scalaires) ; il devient clair alors qu'une compilation efficace d'une telle demande ne sera jamais possible que dans des cas très particuliers.

e) l'opérateur ρ

L'un des principaux moyens qu'a le mathématicien pour définir de "grands" objets (au sens combinatoire) munis d'une structure régulière consiste en des définition "récursives" dont les plus simples ont été codifiées par l'utilisation des symboles Σ et Π ; la syntaxe ternaire (ou pire) et l'écriture bi-dimensionnelle de tels symboles les rend impropre à un traitement automatique aisé. Le problème est résolu partiellement par la plupart des langages de programmation de façon "dynamique" : boucles dq du X Fortran ou de l'Algol par exemple. APL, en accord avec son caractère "statique," préfère utiliser la réduction vectorielle ; les boucles ne sont plus qu'un cas particulier de branchements et tests dans les programmes. Les vecteurs de polynômes n'existant pas en LIMA, il était néanmoins nécessaire de trouver une notation commode. La première idée venant à l'esprit était de faire comme s'ils existaient, et si V désigne une expression "interprétable" comme un vecteur de polynômes, +/V serait la somme de ces polynômes. Restait à définir un système de notation pour V . L'opérateur ρ (dyadique) employé en APL pour structurer (reshape) paraissait s'imposer ,mais la structure d'APL (n ρ P pour le vecteur P,P,...,P) était trop restrictive.

Après divers tatonnements, la syntaxe adoptée finalement fut :

+

ou / $(I_1 \in exp_1) \wedge (I_2 \in exp_2) \wedge \ldots \wedge (I_n \in exp_n)$ ρ exp poly

×

Tout d'abord, un ρ doit donc être précédé d'une réduction : + / ou × / (en fait, d'autres opérateurs peuvant être employés, mais la non commutativité rend le résultat assez aléatoire ...) La signification d'une telle expression est la suivante : la première expression à gauche (de la forme I ∈ exp) est évaluée, et exp doit être un vecteur ; alors I prend pour valeur le premier élément de ce vecteur, et la deuxième expression est à son tour évaluée, etc... Quand on arrive au ρ , l'expression (polynômiale) à sa droite est évaluée, ajoutée (ou multipliée aau résultat provisoire, puis on recule d'un cran, la dernière variable "de contrôle" prend pour valeur le deuxième élément du vecteur qui lui correspond, et ainsi de suite.

Cette description assez embrouillée correspond au fond à une série de boucles do (au sens de l'Algol) emboitées ; voici quelques exemples (rappelons que ⍳N désigne (en APL et en LIMA A) le vecteur 1 2 ... N (ou le vedteur vide si N = 0)

$$+/(I\in\iota 4)\rho X\times I \qquad = \sum_{i=1}^{4} X^i = X^4 + X^2 + X^3 + X$$

$$+/(I\in\iota 5)\wedge(J\in\iota 3)\wedge(K\in\iota J)\rho X[I;J]*K \qquad = \sum_{i=1}^{5}\sum_{j=1}^{3}\sum_{k=1}^{j} X_{i,j}^{k}$$

$$\times/(I\in\iota 0)\rho P \;=\; 1 \quad .$$

La souplesse de cette notation ne doit pas dissimuler son caractère hétérodoxe par rapport au reste de LIMA A, ni les risques d'ambiguïté syntaxique qu'elle provoque. Concernant, ce dernier point, le prix à payer est l'obligation de toutes les parenthèses "séparant" les Λ (qui deviennent ainsi inutiles., mais restent typographiquement commode !), et une analyse syntaxique assez sophistiquée (en particulier la réduction et le ρ fonc-

tionnant comme une paire de "parenthèses" séparant deux (et non plus un) fragment, l'analyse commence par la recherche du ρ le plus à gauche et extérieur à toute parenthèses). L'aspect "hétérodoxe" vient surtout de l'apparition, dans un langage "immédiatement interprétable", d'expressions contenant des variables n'ayant pas encore de valeur : les I_1 , I_2 ... sont des variables provisoires (perdant d'ailleurs leur valeur à la sortie de l'évaluation du ρ , tout comme la variable de controle d'une boucle do) au fond, une instruction de ce type est déjà un sous-programme à elle seule, et il s'agit là d'un mélange (à manipuler avec précaution) des deux modes principaux d'APL .

Le mode définition.

On désigne ainsi, en LIMA A comme APL, un mode où l'utilisateur entre des textes qui ne seront utilisés qu'ultérieurement : textes de fonctions et sous-programmes en APL, mais aussi listes de règles de réduction en LIMA A .

La définition des sous-programmes en LIMA A se fait comme en APL, mais les "noms" de fonctions sont limités à la forme Φn ; et les arguments (il s'agit ici des arguments "formels") doivent obligatoirement être P,Q ou R ; c'est là un des nombreux inconvénients de la difficulté de manipuler des noms en APL (difficulté qui disparaîtra quand l'opérateur exécute sera commercialisé), et auquel on aurait pu remédier dans une version non expérimentale de LIMA A . Toutefois, la récursivité a été maintenue ; il suffit en effet de se servir de celle d'APL ! (Un autre problème lié à l'absence de noms est l'impossibilité d'utiliser des étiquettes : le " go to" → est donc purement numérique).

Les règles de substitution sont introduites par le signe Δ (l'entrée dans le mode définition des fonctions se faisant par ∇). Dans le cas le plus simple, une règle est de la forme [n] M → P (où n est un entier explicité, et M un monôme) ; mais on peut aussi introduire une lis de règles par une déclaration $n_1 \leqslant I \leqslant n_2$ (où n_1 et n_2 sont des entiers explicités et I un nom de variable polynômiale non utilisé par ailleurs (sinon son contenu sera détruit lors de l'application de la règle).

Cette déclaration doit être suivie par une ligne de syntaxe [I] M→P .

Une fois une règle entrée, le système ne l'analyse pas immédiatement, mais se contente de mettre à jour le tableau des numéros des règles ; lors d'un appel ultérieur, si besoin en est, la lettre de contrôle se verra affectée, et la règle sera analysée, puis la substitution effectuée, c'est à ce stade seulement que les erreurs éventuelles seront diagnostiquées

Un système de correction similaire à celui d'APL (mais plus fruste) existe tant pour les fonctions que pour les règles.

Voici donc comment les règles de l'introduction pourraient être rentrées dans le système : Δ 1≤I≤256

```
[I] X[I]*2 →+/(J∈ιI-1)ρ<A[I],A[J]>X[J]
Δ
```

(Ceci supposant que la syntaxe interne des < , > corresponde à celle de LIMA A ; on pourrait aussi employer.

```
[I] X[I]*2 → +/(J∈ιI-1)ρ(A[I] ϕ4 A[J])X[J]
```

et définir par ailleurs le produit scalaire Φ 4).

La conception de la notion de règle en LIMA A est en fait une tentative de répondre rapidement à des besoins pratiques, mais masque un problème théorique : que veut dire "simplifier" par la règle $M \to P$?

Si les variables figurant dans M ne figure pas dans P, où y figurent avec un degré plus petit, alors, on arrivera tôt ou tard à un polynôme "réduit", représentant "canonique" du polynôme initial dans la structure quatient (mais ce n'est déjà plus vrai s'il y a plusieurs règles, par exemple $X \to Y^2$ et $Y \to X^2$!). Mais que faire d'une règle comme $X \to X^3$?

Force est de reconnaître qu'il y a une différence entre relation d'équivalence engendrée par une équation et ordre de simplification ou de substitution.

Une partie du problème sera abordée dans un LIMA ultérieur (LIMA Φ), mais on sait [4] [5] que le problème général est récursivement insoluble dans la plupart des cas pratiques intéressants. Des solutions pragmatiques du genre de celle que nous proposons devront donc être adoptées : en LIMA A, la solution retenue est de laisser le mathématicien faire lui-même presque toutes les "simplifications" ; en LIMA Φ, le travail automatique de l'ordinateur sera interrompu périodiquement pour réclamer des indications de méthode à l'utilisateur.

Les commandes-système.

Tout d'abord, des commandes analogues à celles d'APL se contentent de réclamer des informations sur l'état du système : ce sont)VARS,)FNS, et)RULES produisant respectivement la liste des variables (polynômiales), des fonctions et des numéros de règles).

)VNAMES, donnant la liste des lettres permises comme noms de variables.

Comme en APL , d'autres commandes changent l'état du système, mais les informations que nécessite LIMA A pour ce faire donnent alors naissance à un dialogue :)MODE n (n = 1,2,3 ou 4) fait passer tout le système dans le mode correspondant, et réclame, pour les modes 2 et 4 , l'information WIDTH ? (nombre maximum d'indéterminées par monôme)*, pour les modes 3 et 4 l'information CODE ? (valeur maximum que peuvent prendre exposants et indices). Il est possible de spécifier après la commande MODE une liste de variables ; ne seront converties que celles-là (et dans le dialogue, la demande DROP ? effacera les autres ou les maintiendra). Après le dialogue, le système renvoie WAS MODE n'.

(En fait, l'effet de ces commandes est invisible pour l'utilisateur, mais il constatera en général d'importants gains d'espace (en modes 3 et 4) ou de temps (en mode 1) ; parfois, le système n'acceptera de fonctionner que dans certains modes (par exemple, si le nombre de variables est très grand, seulement en mode 2)).

Le type des constantes (c'est-à-dire l'anneau A) est donné par la commande) TYPE ; sont actuellement admis) TYPE REAL, COMPLEX , Z/n (où n est un entier explicité) et naturellement USER n .....

L'entrée dans les modes USER s'accompagne d'un dialogue réclamant quelques informations sur le format interne des constantes ; en général, à ce stade l'utilisateur devra sortir de LIMA A , et aller donner, en APL des détails sur les calculs dans A ; pour se faire, la commande → APL provoque une interruption ; le retour en LIMA A se fait (en APL !) par →LIMA!

Par contre, les erreurs (syntaxiques et autres) ne font pas sortir de LIMA ; pour conclure une séance, on utilise la commande)OFF .

L'initiation des différents paramètres du système se fait grâce à la commande) CLEAR ; ce sera en général le moment choisi par l'utilisateur pour spécifier les noms des indéterminées et des constantes. Les commandes)CNAMES et) INAMES respectivement impriment la liste des constantes (de A et des indéterminées. Puis le système imprime MORE ?= et les noms fournis par l'utilisateur à ce stade correspondent aux déclarations de tableaux du Fortran : Y(2 ; 3) annonce les six indéterminées $Y_{1,1}$; $Y_{1,2}$; $Y_{1,3}$ $Y_{2,1}$; $Y_{2,2}$; $Y_{2,3}$ lesquelles viennent (du point de vue de l'ordre lexicographique) après les indéterminées précédemment introduites.

Bien que la pratique mathématique usuelle tolère l'utilisation simultanée de X et de X_1 , elle a été interdite en LIMA A .

V. CONCLUSIONS.

Comme on l'a dit dans l'introduction, le but essentiel de LIMA A était un certain nombre d'expériences linguistiques ; dans un système plus général, beaucoup de ces essais (substitutions, vecteurs de noms ,...) devront être remplacés par des traitements plus systèmatiques, soumis à une syntaxe uniforme. Le rôle de l'opérateur (sous sa forme actuelle devra naturellement être repensé ; toutefois, il semble qu'il y ait beaucoup d'avenir à des notations opératoires très puissantes de ce type.

Du point de vue algèbrique proprement dit, LIMA A n'est qu'un début ; par contre, les besoins courants de manipulations polynômiales semblent

satisfaits ; à titre d'exemple, nous nous proposons d'écrire en LIMA A des programmes de calcul des polynômes de Matijasevic [6] correspondant à des relations récursives données ; en dépit du ralentissement dû au passage dans deux interpréteurs, les temps de calcul restent meilleurs qu'à la main.

Toutefois, à ce sujet, il devient clair qu'intéresser le mathématicien à ce genre de recherches nécessites qu'on lui fournisse un outil opérationnel le passage à l'écriture en langage-machine au moins des sections les plus usitées de LIMA A devient donc nécessaire.

BIBLIOGRAPHIE

[I] P.BRAFFORT Note ECSTASM n° I: le projet LIMA Orsay I974

[2] N.BOURBAKI Elements de Matehematique, Algèbre, Chapitre 4
Hermann I967

[3] R.GHANDOUR et A. MEZEI General arrays, operators and functions
IBM J. of Research I7 335 (I973)

[4] P.S.NOVIKOFF On the algorithmic unsolvability of the word problem in group theory Moscou I955

[5] D.RICHARDSON Some unsolvable problems involving elementary functions of a real variable
J.of Symbolic Logic 33 5I4

[6] Y.V. MATIJASEVIC On the recursive unsolvability of Hilbert's tenth problem
Proc. of the 4 th Int. congress for Logic
North-Holland I973

Annexe 1 : UNE SESSION

Initialisation du Système.

```
      LIMAA
)CLEAR
```

On définit 5 indéterminées : X , Y , Z_1 , Z_2 , Z_3 .

```
)INAMES

MORE?
X Y Z(3)
```

Le mode de représentation interne est fixé.

```
)MODE 1
WAS MODE 1
```

Les constantes seront des nombres complexes

```
)TYPE COMPLEX
WAS Z/3
```

$(X+iY)(X-iY) = X^2+Y^2$.

```
(X+<ι>Y)X-<ι>Y
 2    2
X  + Y
```

Utilisation du symbole ρ : $\sum_{i=1}^{3} Z_i$ (noté +/(I ∈ ι3) ρ Z[I]) .

Le résultat est affecté à la variable P , et montré grâce au quad □ .

```
□←P←+/(I∈ι3)ρZ[I]
Z  + Z  + Z
 1    2    3
```

Calcul de $\prod_{i=1}^{6} X^{i}$.

```
×/(Iε⍳6)ρX*I
```

X^{21}

Calcul de $P^3 = (Z_1 + Z_2 + Z_3)^3$.

```
□←Q←P*3
```

$$Z_1^3 + 3Z_1^2 Z_2 + 3Z_1^2 Z_3 + 3Z_1^2 Z_2 + 6Z_1 Z_2 Z_3 + 3Z_1 Z_3^2 + Z_2^3 + 3Z_2^2 Z_3 + 3Z_2 Z_3^2 + Z_3^3$$

Calcul de $(P^3)''_{Z_1,Z_2} = \frac{\partial^2 (P^3)}{\partial Z_1 \partial Z_2}$.

```
[Z[1];Z[2]]'Q
```

$6Z_1 + 6Z_2 + 6Z_3$

Substitution simple : X remplace Z_1 et Z_3 remplace Z_2 dans P .

```
P[X;Z[3];Z[3]]
```

$X + 2Z_3$

G_n entre la liste de règles : $Z_3 \rightarrow Z_2^2$; $Z_2 \rightarrow Z_1^2$; $Z_1 \rightarrow Z_o^2$ (à ce stade, Z_o n'est pas défini).

```
∇ ENTREE D'UNE LISTE DE REGLES DE SIMPLIFICATION
Δ1≤I≤3
[I]Z[I]→Z[I-1]*2
```

Z_o est défini comme nouvelle indéterminée.

```
)INAMES
X     Y     Z[I](1≤I≤3)
MORE?
Z[0]
```

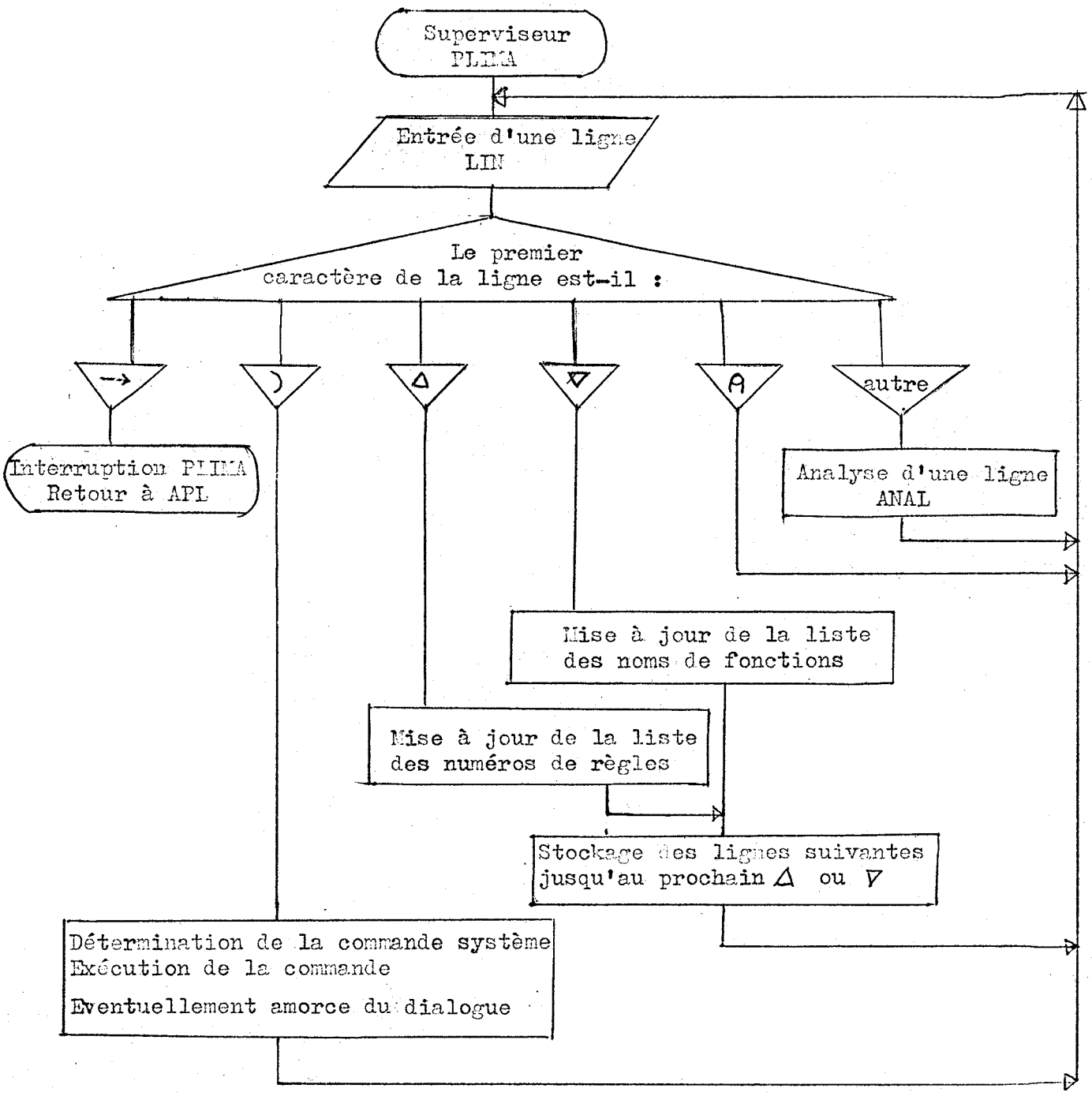
2 tentatives de simplification : les règles sont appliquées dans l'ordre (3,2,1) (qui donne le meilleur résultat), puis 1,2,3.

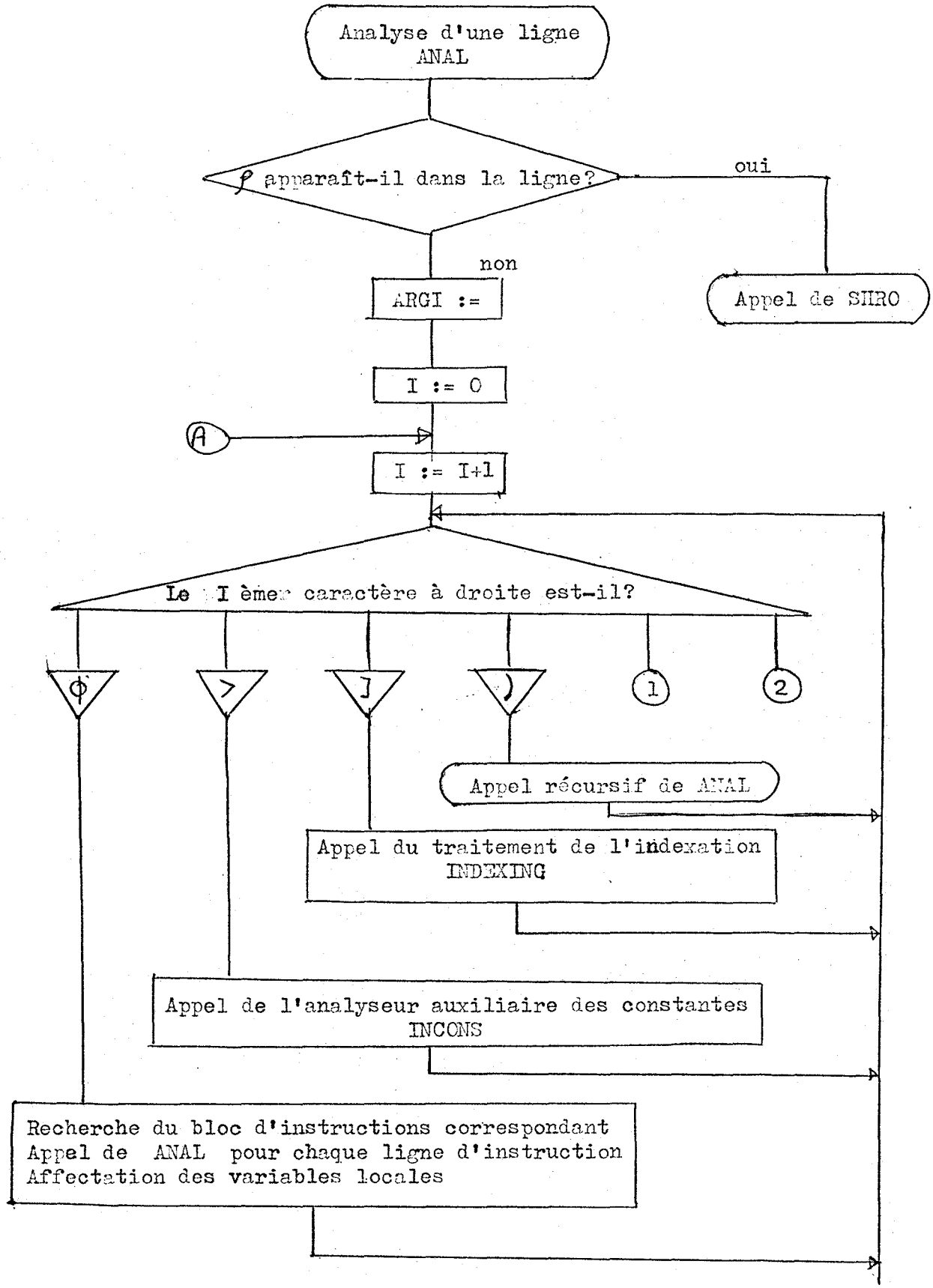
```
3 2 1/P
  8     4     2
Z   + Z   + Z
 0     0     0
1 2 3/P
 2     2     2
Z   + Z   + Z
 0     1     2
)OFF
```

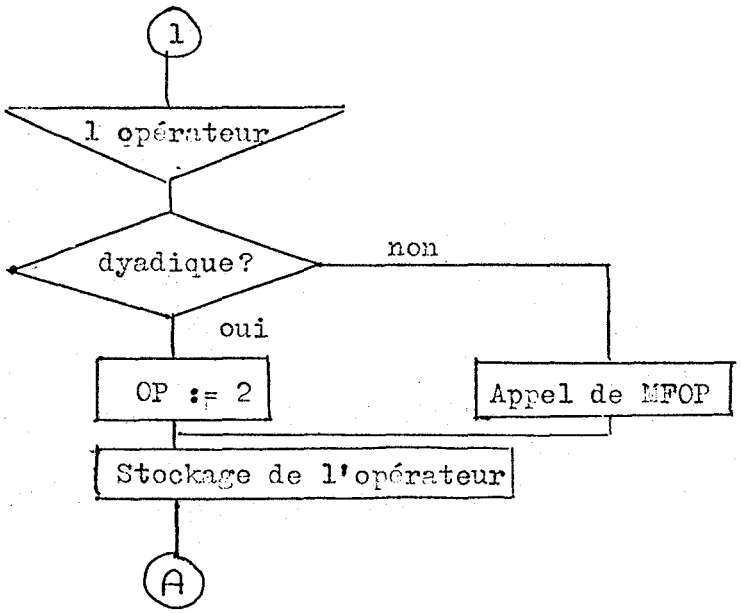
Appendice 2 : Organisation générale de LIMA A

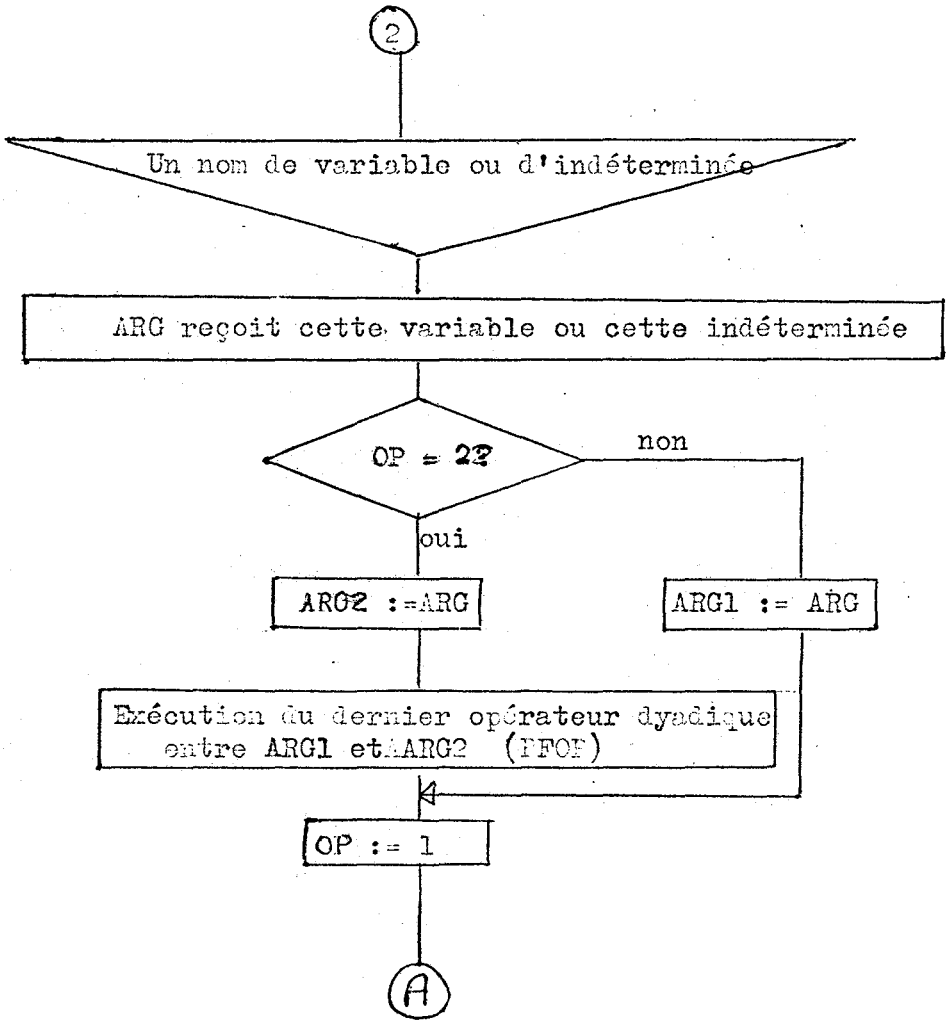
Appendice 2

Organisation générale de LIMA A




Analyse d'une ligne
ANAL
ρ apparaît-il dans la ligne?
oui
non
Appel de SHRO
ARGI :=
I := 0
A
I := I+1
Le I ème caractère à droite est-il?
Φ
>
J
)
1
2
Appel récursif de ANAL
Appel du traitement de l'indexation
INDEXING
Appel de l'analyseur auxiliaire des constantes
INCONS
Recherche du bloc d'instructions correspondant
Appel de ANAL pour chaque ligne d'instruction
Affectation des variables locales


1
1 opérateur
dyadique?
non
oui
OP := 2
Appel de MFOP
Stockage de l'opérateur
A


2
Un nom de variable ou d'indéterminée
ARG reçoit cette variable ou cette indéterminée
OP = 2?
non
oui
ARG2 :=ARG
ARG1 := ARG
Exécution du dernier opérateur dyadique entre ARG1 et ARG2 (FFOP)
OP := 1
A

Apendice 3 Algunos programas - comentados

```
      ∇PLIMA[□]∇
    ∇ PLIMA
[1]  IN:ERR←0
[2]   V1ERR←⍳0
[3]  IN:LIN←LIGN←INPUT
[4]   →(LIN[1]=')ΒΔV→')/SYS,IN,RUL,PMON,APL
[5]   TRES←0
[6]   NHIER←HIER LIN
[7]   ANAL LIN
[8]   →(ERR≠1)/TPR2
[9]   ERR←0
[10]  →((⍴V1ERR)=0)/LO
[11]  V1ERR
[12]  V1ERR←⍳0
[13] LO:LIGN
[14]  ((0⌈CCAR-1)⍴' '),'+'
[15]  →((⍴V2ERR)=0)/IN
[16]  V2ERR
[17]  →IN,V2ERR←⍳0
[18] TPR2:→(LAP=1)/IN
[19]  →(CRES= 0 1 2 3)/POL,ENT,VENT,CONS
[20] POL:T0←TRES
[21]  H0←HRES
[22]  PPRINT
[23]  →IN
[24] ENT:OUTT TRES
[25]  →IN
[26] VENT:I←⍳J←0
[27] BO:→((J←J+1)>⍴TRES)/FI
[28]  I←I,(OUTT TRES[J]),'  '
[29]  →BO
[30] FI:I
[31]  →IN
[32] CONS:OUTPUT TRES
[33]  →IN
[34] RUL:::
[35]  FON←LIN←1↓CBL LIN
[36]  VF← 0 0 0
[37] TSY:→(LIN[1]='ΡΩΦ')/ERT,NEDY,NEMO
[38] ERROR:'SYNTAX ERROR'
[39]  →IN
[40] ERT:→(LIN[2]≠'←')/ERROR
[41]  VF[1]←1
[42]  LIN←2↓LIN
[43]  →(LIN[1]='Ρ')/ERROR
[44]  →TSY
[45] NEDY:→(LIN[2]≠'Φ')/ERROR
[46]  LIN←1↓LIN
[47]  VF[2]←1
[48] NEMO:→(LIN[⍴LIN]≠'Ρ')/PMON
[49]  LIN←1↓¯1↓LIN
[50]  VF[3]←1
[51]  →PMON+1
[52] PMON:LIN←1↓LIN
[53]  FN←INTPUT LIN
[54]  →(ERR=1)/ERROR
[55]  →(∧/VF[2 3]= 1 0)/ERROR
[56]  SYNT←SYNT,VF
[57]  VNF←VNF,FN
[58]  LPHI←LPHI,PHI←⍴NABVEC
[59]  NABSTOCK
[60]  →IN
[61] APL:→((⍴LIN)≠4)/ERSY
[62]  →(∧/LIN='→APL')/AP
[63] ERSY:'SYNTAX ERROR'
[64]  →IN
[65] AP:'AFTER THE ''INTERRUPT'' MESSAGE,YOU WILL BE IN APL INTERRUPT MODE,RETURN IS BY →LIMA'
[66]  .
[67] SYS:K←LIN⍳' '
[68]  CVPA←1↓(K-1)↑LIN
[69]  LIN←K↓LIN
[70]  →(LITC∧.=LL↑CVPA)/0,TYPAC,CLEC,VARC,FNSC
[71]  'INCORRECT COMMAND'
[72]  →IN
[73] TYPAC:K←((LIN=' ')∨LIN='/')⍳1
[74]  COPA←(K-1)↑LIN
[75]  SPA←K↓LIN
[76]  →(LITC∧.=LL↑CVPA)/TYPAC,CLEC,VARC,USAC
[77]  'UNKNOWN TYPE'
[78]  →IN
[79] REAC:OPT
[80]  →IN,OPTION←TYPE←1
```

PLIMA (suite)

```
[81] FNSC:W←⍳I←0
[82] BP:→((I←I+1)>⍴VNF)/FP
[83]  W←W,'Φ',(OUT VNF[I]),'
[84]  →BP
[85] FP:→IN,⍴□←W
[86] COAC:OPT
[87]  →IN,OPTION←TYPE←2
[88] ZNAC:NM1←INTPUT SPA
[89]  →(ERR=1)/USAC+1
[90]  OPT
[91]  →IN,OPTION←2+TYPE←1
[92] USAC:OPTIO←3+INTPUT SPA
[93]  →(ERR=0)/SUS
[94]  V1ERR
[95]  LIN
[96]  14⍴' ','+'
[97]  →IN
[98] CLEC:VNAL←29⍴0
[99]  NABVEC←''
[100] VLITN←,0
[101] VNF←LPHI←⍳0
[102] SYNT← 3 0 ⍴0
[103] →IN
[104]SUS:OPT
[105] →IN
[106]VARC:VVV←VNAL/ALPH
[107] ((2×⍴VVV)⍴ 0 1)\VVV
[108] →IN
      ∇
```

```
     ∇ANAL[⎕]∇
   ∇ ANAL V;N;CODE;K;K11;TVAL1;TVAL2;HVAL1;HVAL2;C1;C2;DECAL;I
     →(ERR=1)/LAF←0
     V←,V
     →((ρV)≠0)/SU
     TRES←⍳0
     CRES←HRES←2
     →0
   SU:DECAL←TRES
     N←ρV
     →(~'ρ'∊V)/12+CODE←1
     SHRO V
     →(ERR=1)/0
     →(N=0)/FIN
   DEB:→LABEL[PERM⍳V[N]]
   ERR1:V1ERR←'SYNTAX ERROR'
     ERR
     →0
   LETTRE:BIT←1
     CAR←V[N]
     INDEXING
     DFOP
     →BOUCLE
   CHIFFRE:VNUM←N NSEGMCH V
     N←UTR
     →(V[N]='-⌽')/MIN,FONCT
     →(V[N]∊CALP,VALP)/IND
     TVAL2←VNUM
     C2←HVAL2←1
     DFOP
     →BOUCLE
   MIN:TVAL2←-VNUM
     →MIN-3
   IND:→(1≠DIM[K←(CALP,VALP)⍳V[N]])/ERR1
     →(VNUM>VDIM[BV[K]])/ERR2
     INDEXING
     →BOUCLE
   FONCT:→(~(K11←VNUM)∊VFN)/ERR3
     →(SYNT[IF←VNF⍳Z;2]=1)/FMN
     →BOUCLE,ρCODE←8
   FSEP:N←(K←N) OSEP V
     TVAL2←INCONS N↓(K-1)↑V
     →(ERR=1)/0
     C2←HVAL2←3
     DFOP
     →BOUCLE
   MONADIC:MFOP V[N]
     →(ERR=1)/0
     →BOUCLE
   FPARENT:N←(K←N) OPAR V
     TRES←N+DECAL
     ANAL N↓(K-1)↑V
     →(ERR=1)/0
     TVAL2←TRES
     HVAL2←HRES
     C2←CRES
     DFOP
     →BOUCLE
   INQU:'⎕:'
     TRES←N+DECAL
     ANAL ⍞
     →FPARENT+2
     →BOUCLE
   DYADIC:CODE←CODE⍳K12←V[N]
     →(K12∊'/⍉''')/EVC
   BOUCLE:→(0=N←N-1)/FIN
     →(ERR=1)/0
     →DEB
   OUTQU:STEC
     CODE←1+LAF←1
     →BOUCLE
   AFFECT:→(N=1)/ERR
     →(V[N-1]='⎕')/AFQU
     →(V[N-1]=']')/IND
     →((K11←ALPH⍳K←V[N-1])=30)/ERR
     →(~K∊EALP)/ERS
     T0←TVAL1
     H0←HVAL1
     TSAL K11
     LAF←1
     N←N-1
     CODE←0
```

ANAL (suite)

```
[81]  →BOUCLE
[82] ERR:ERR
[83]  V1ERR←'AFFECTATION ERROR'
[84]  →0
[85]  →
[86] FIN:→(CODE>3)/ERS
[87]  TRES←TVAL1
[88]  HRES←HVAL1
[89]  →0,CRES←C1
[90] ERS:ERR
[91]  V1ERR←'SYNTAX ERROR'
[92]  →0
[93] AFQU:N←N-1
[94]  →OUTQU
[95] IND:N←(K←N) OPCR V
[96]  →(~V[N]∈EALP)/ERRAF
[97]  INDEXING
[98]  →(C2=1)/ENT
[99]  →(C2≠3)/ERRDOM
[100] TVAL2 AFIND KN
[101] →BOUCLE
[102]ENT:(SHAPE TVAL2) AFIND KN
[103] →BOUCLE
[104]ERRAF:?
[105]ERRDOM:ERR
[106] V1ERR←'DOMAIN ERROR'
[107] →0
[108]CRO:BIT←0
[109] INDEXING
[110] DFOP
[111] →BOUCLE
[112]EVC:→(V[N-1]≠']')/BOUCLE
[113] →(K12='/⍵''')/SUBS,CALVEC,CALVEC
[114]SUBS:N←(K←N-1) OPCR V
[115] W←N↓(K-1)↑V
[116] →(~'→'∈W)/ERRSUB
[117] W1←(K←W⍳'→')↑W
[118] W2←(K-1)↑W
[119] TRES←N+DECAL
[120] ANAL W2
[121] →(ERR=1)/0
[122] →TESTMONOME
[123]MONO:TRES←N+DECAL+K
[124] ANAL W1
[125] →(ERR=1)/0
[126] →(CRES≠0)/NNPOL
[127]INDEX:BEX← 1 0 0 0
[128] P1←TRES
[129] K1←HRES
[130] T5←TVAL1
[131] H5←HVAL1
[132] SUBST VS
[133] TVAL1←T6
[134] HVAL1←H6
[135] CODE←2
[136] →BOUCLE
[137]NNPOL:→(CRES= 1 2 3)/EE,ERRDOM,CCC
[138]EE:ERES←SHAPE TRES
[139] TRES←(WIDTH,1)⍴0
[140] →INDEX
[141]CCC:MPES←TRES
[142] →EE+1
[143]ERRM:ERR
[144] V1ERR←'MUST BE A MONOMIAL'
[145] →0
[146]CALVEC:::
[147]ERRSUB:ERR
[148] V1ERR←'SYNTAX ERROR'
      ∇
```

```
      ∇DFOP[□]∇
    ∇ DFOP
[1]   →(CODE≠1)/SUI1+LAF←0
[2]   TVAL1←TVAL2
[3]   HVAL1←HVAL2
[4]   C1←C2
[5]   →FIN
[6]  SUI1:→(CODE∈SPEOP)/SUI2
[7]   TYPE←TY[C1+1;C2+1]
[8]   →CONV[C1+1;C2+1]
[9]  CE1:HVAL1←SHAPE TVAL1
[10]  TVAL1←(WIDTH,1)ρ0
[11]  →SUI2
[12] CE2:HVAL2←SHAPE TVAL2
[13]  TVAL2←(WIDTH,1)ρ0
[14]  →SUI2
[15] CC1:HVAL1←(TYPE,1)ρTVAL1
[16]  TVAL1←(WIDTH,1)ρ0
[17]  →SUI2
[18] CC2:HVAL2←(TYPE,1)ρTVAL2
[19]  TVAL2←(WIDTH,1)ρ0
[20]  →SUI2
[21] CCE2:TVAL1←SHAPE TVAL1
[22]  →SUI2
[23] CCE1:TVAL2←SHAPE TVAL2
[24]  →SUI2
[25] ERRC:ERR
[26]  V1ERR←'DOMAIN ERROR'
[27]  →0
[28] SUI2:→DFAD[CODE]
[29] MULT:→MUL[TYPE]
[30] POL1:GTR ¯7 2 ¯8 1
[31]  PMULT
[32]  GTR ¯3 7
[33]  →FIN,C1←0
[34] ENT1:TVAL1←TVAL2×TVAL1
[35] RET:C1←HVAL1←HVAL2⌈HVAL1
[36]  →FIN
[37] CONS1:TVAL1←TVAL2 FOIS TVAL1
[38]  →FIN,C1←3
[39] ADD:→ADAD[TYRE]
[40] POL2:GTR ¯7 2 ¯8 1
[41]  PADD
[42]  TS 7
[43]  →FIN,C1←0
[44] ENT2:TVAL1←TVAL2+TVAL1
[45]  →RET
[46] CONS2:TVAL1←TVAL2 PLUS TVAL1
[47]  →FIN,C1←3
[48] EXP1:→(C1≠1)/PV
[49]  →EXPAD[C2+1]
[50] POL3:GTR ¯8 3
[51]  ST 1
[52]  EXP TVAL1
[53]  GTR ¯3 7
[54]  →FIN,C1←0
[55] PV:→((C2≠1)∨C1≠2)/ERRC
[56] ENT3:TVAL1←TVAL2*TVAL1
[57]  →RET
[58] CONS3:TVAL1←TVAL1 PEXP TVAL2
[59]  C1←HVAL1←HVAL2
[60]  →FIN,C1←3
[61] REGL:→(∨/C2≠ 1 2)/ERRS
[62]  →(C1=2)/ERRS
[63]  →(C1=0)/FIN
[64]  VR←,TVAL2
[65]  I←0
[66] BO:→((I←I+1)ρV)/FIN
[67]  REXEC VR[I]
[68]  GTR ¯7 5
[69]  SUBST VRUL
[70]  GTR ¯6 7
[71]  →BO
[72] FIN:→0,CODE←2
    ∇
```

```
      ∇SHRO[□]∇
    ∇ SHRO V;I;W;W1;V1;V2;CC;N;MO;NEXP;NAL;WW;PA;TPR;CODE;CDE
[1]   I←0
[2]   BO:→((I←I+1)>ρV)/0
[3]    →((V[I]≠'ρ')∨NHIER[I+DECAL]≠NHIER[DECAL+1])/BO
[4]    W1←I↓V
[5]    N←I
[6]    V1←⍬V2←(I-1)↑V
[7]   LOOK:N←V1⍳'/'
[8]    →(N>ρV1)/ERR1
[9]    →(∨/(CC←V1[N+1])='×')/OUI
[10]   V1←N↓V1
[11]   →LOOK
[12]  OUI:N←(ρV1)-N
[13]   V2←(N+1)↓V2
[14]   →((ρV2)=0)/ERR2
[15]   DEC←N-1+(ρV2)+I←0
[16]   MO←0
[17]  BO2:→((I←I+1)>ρV2)/FI2
[18]   →((V2[I]≠'∧')∨NHIER[I+DEC]≠NHIER[DEC])/BO2
[19]   MO←MO,I
[20]   →BO2
[21]  FI2:MO←MO,I
[22]   NEXP←¯1+ρMO
[23]   W←V2≠'('
[24]   NAL←⍳I←0
[25]  BO3:→((I←I+1)>NEXP)/FI3
[26]   WW←MO[I]↓W
[27]   PA←MO[I]+WW⍳1
[28]   →(~V2[PA]∈EALP)/ER2
[29]   →BO3,NAL←NAL,ALPH⍳V2[PA]
[30]  ER2:ERR
[31]   V1ERR←'NOT A VARIABLE'
[32]   →0
[33]  FI3:→INIT
[34]  INIT:→(CC='+')/ADD
[35]   C1←TVAL1←HVAL1←¯1+CDE←2
[36]   →SU
[37]  ADD:CDE←2+C1←HVAL1←1+TVAL1←0
[38]  SU:TPR←NEXPρI←0
[39]  RECANAL:→((I←I+1)>NEXP)/CALC
[40]   W←MO[I]↓(MO[I+1]-1)↓V2
[41]   PA←(W≠'(')⍳1
[42]   W←((PA-1)↑W),(PA+1)↓W
[43]   TRES←MO[I]+DECAL
[44]   ANAL W
[45]   →(ERR=1)/0
[46]   →((CRES=1)∨CRES=2)/OUJ
[47]  ERR1:ERR
[48]   V1ERR←'DOMAIN ERROR'
[49]   →0
[50]  OUJ:T0←,TRES
[51]   TPR[I]←VNAL[NAL[I]]
[52]   H0←10
[53]   →GDOWN+1
[54]  GDOWN:TRAL NAL[I]
[55]   →((H0-10)≥ρT0)/DOW
[56]   H0←H0+1
[57]   TSAL NAL[I]
[58]   →RECANAL
[59]  DOW:VNAL[NAL[I]]←TPR[I]
[60]   →((I←I-1)=0)/FIN
[61]   →GDOWN
[62]  CALC:TRES←N+DECAL
[63]   ANAL W1
[64]   →(ERR=1)/0
[65]   TVAL2←TRES
[66]   HVAL2←HRES
[67]   C2←CRES
[68]   CODE←CDE
[69]   DFOP
[70]   →DOW+1
[71]  ERR:N←N-1
[72]   CODE←2
    ∇
```

PLIMA

[5] à [33] : Traitement des lignes en mode éxécution, impression éventuelle des diagnostics d'erreur([9] à [17]) et du résultat([18] à [33]).

[34] à [60] : Traitement de Δ et ∇ ; les lignes [37] à [50] analysent la structure du "header"(de type $\underline{P} \leftarrow \underline{Q} \phi n \underline{R}$).

[61] : Interruption(→APL):Aprés celle-ci, la commande → LIMA (en APL) ramène à [1] dans PLIMA .

[70] : Le produit interne $\wedge.=$ compare la commande-système analysée aux lignes de LITC, formée des commandes permises.

ANAL

[13] ≠ Aiguillage dépendant du I^{ème} caractère; LABEL contient les adresses des différentes étiquettes.

[19] ≠ INDEXING est un programme technique traitant les expressions entre crochets.

[48]: OPAR repère la parenthèse ouverte correspondant à celle (fermée) qui vient d'être analysée.

[70] à [82] : Traitement de la flèche d'affectation.

[114] à [148] : Traitement des substitutions; les lignes [128] à [134] manipulent les registres internes discutés dans DFOP.

DFOP

Opération dyadique entre les deux arguments ARG1 et ARG2(placés dans les qyatre "registres" TVAL1,HVAL1, et TVAL2,HVAL2).

[2] à [5] :Il n'y a pas d'opérateur: transfert de ARG2 à ARG1.

[6] à [24] :Conversion de type en cas d'incompatibilité entre ARG1 et ARG2.

[28] : Aiguillage, dépendant de l'opérateur.

[29] à [38] ≠: Multiplication.

[30] ≠ Première apparition de GTR: en fait, les opérations sont effectuées dans 8 registres internes;GTR effectue des transfert entre ces registres(ainsi que les programmes ST et TS).

[39] à [47] :Addition.

[48] à [60] :Exponentiation.

[61] à [72] ≠ Substitutions.

SHRO

Analyse des lignes contenant le symbole ρ

[3] : NHIER repère les niveaux de parenthésage de la ligne analysée, le but de [3] est l'extraction du premier ρ (à gauche) non parenthésé.

[7] à [11] : Recherche du / correspondant(précédé par + ou x).

[17] à [33] : Décomposition entre / et ρ de la ligne en éléments de la forme (I ∈ ....)

[34] à [72]: Calcul effectif de l'expression.

APL SYMBOL PROCESSING IN APL PROGRAM VERIFICATION

W. VERVOORT

1. The assertion method of FLOYD

2. ISVAP - An interactive system for verification of APL programs

3. Automatic propagation in ISVAP

4. Symbol processing in ISVAP

5. The use of APL

* Technical University

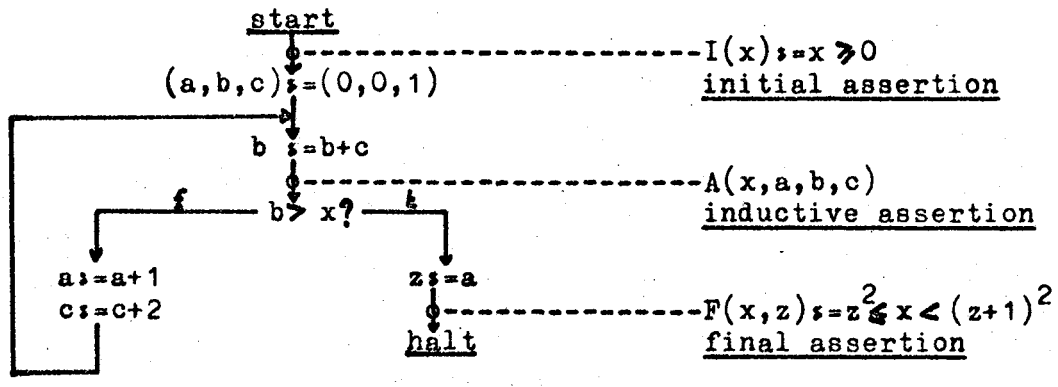
TWENTE

1. The assertionmethod of Floyd

1.1 Example (from Manna)

Consider the following problem:
Find the largest integer z such that $z \leq \sqrt{x}$
The following program P could be a solution:

P:



Prove P to be correct with respect to I(x) and F(x,z).

Proof: P is correct with respect to I and F iff there exists an inductive assertion A(x,a,b,c) such that the following verification conditions (vc's) hold for all x,a,b and c:

(1) $I(x) \supset A(x,0,1,1)$ and

(2) $(A(x,a,b,c) \wedge (b \leq x)) \supset A(x,a+1,b+c+2,c+2)$ and

(3) $(A(x,a,b,c) \wedge (b > x)) \supset F(x,a)$

There is an A(x,a,b,c) such that (1),(2), and (3) are true:

$A(x,a,b,c) := (a^2 \leq x) \wedge (b=(a+1)^2) \wedge (c=2a+1)$.

1.2 Propagation and verification conditions (vc's)

The vc's (1),(2) and (3) are obtained by propagation of one assertion bottom to top (against execution direction) to another through one of the paths of the program.

Example: A(x,a,b,c) propagated through b:=b+c gives: A(x,a,b+c,c), which propagated through (a,b,c):=(0,0,1) gives: A(x,0,1,1).

As one can see substitution is an important primitive action.

After propagation to a point where two assertions meet each other the vc is composed by writing

(1) $I(x) \supset A(x,0,1,1)$

This vc (and the same for the other two) has to be proved true for all x, once an appropriate A has been established.

With ISVAP one can interactivly propagate assertions through the paths of an APL program, and compose vc's.
The vc's are not proved by ISVAP.

In the remaining part I will go a bit more into details of ISVAP in order to find out to what extent symbol processing has been used. This will finally result in some conclusions about the use of APL.

1.3 Conclusions about the method of Floyd

Before going to ISVAP I will at first make some remarks about the assertion method of Floyd itself.

1. ISVAP expects inductive assertions to be specified before the propagation has started. It is my experience that this is not always an easy job, when an existing program has to be proved:

   How can one formulate A(x,a,b,c) if the vc's (1),(2) and (3) are not yet formulated!

Manna gave some heuristic methods to find A by considering the assignments in the loop and constructing invariants from recursive equations. I believe one should never attempt to make a program without knowing A(x,a,b,c).

2. With Floyd one proves "partial correctness" (Manna);
One proves a program to be correct if it stops, but one does not prove it to stop. One has to show this separately, and this can easily be done in the example.

2. ISVAP- An Interactive System for Verification of Apl Programs

2.1 Programs of ISVAP

ISVAP consists of a couple of programs, which can best be introduced by the following example:

a) Initialization and Input

INITIALIZE	-initialization of internal data
LOAD '<program P>'	-input of programstring P
ANALIZE	-find places for inductive assertions
FINDPATHS <places selected>	-find paths for places selected
I←INASS	-type in Initial assertion
A←INASS	-type in inductive assertion
F←INASS	-type in Final assertion

Assertions are predicate calculus formulas given in APL.

b) Propagation

A1←A PROPAGATE <linenrs. of path in reversed order>	-after each step the propagated assertion is printed and the user can make changes with:
AS←CHANGEASS AS	-editing or substitution by user

c) Composition of vc

VC1←I IMPLIES A1	-(1) $I(x) \Rightarrow A(x,0,1,1)$, with A filled in.

d) Verification of vc

OUTASS VC1	-lists the vc VC1
VC1←CHANGEASS VC1	-editing or substitution of VC1 in order to prove it true

2.2 Storage capacity and speed

Storage capacity

28.8 Kbytes are used by ISVAP programs, without interface variables and space for execution.
Workspacesize should be 72K or three of 32K:
one for initialization,loading and analizing;
one for propagation and
one for verification.
The interface variables are grouped in VAR

Speed

The program is rather slow: about 2 minutes terminal time per propagation step.

3. Automatic propagation

3.1 ASSIGN A←B or X[V]←B

case 1: A←B

Substitute B for A in ASS (the assertion to be propagated)

case 2: X[V]←B V is a vector

If ASS contains X[V] with the same index V then substitute B for X[V] in ASS, and if B contains X then first replace X by X'.

The (remaining) occurences of X (not X') are replaced by (X[V'],B)[⍋V',V] with V':=(~(⍳⍴X)∈V)/⍳⍴X.(all indices not in V).

Finaly X' is replaced by X again.

case 3: X[M]←B B is a matrix, is also implemanted.

3.2 TEST →(C)/L

In the following program scheme:

[i] →(C)/j

[i+1] ---

•

•

[j] ---

there are two paths:	Propagated assertions:
path 1: ....,i,j,...	C⊃ASS
path 2: ....,i,i+1,...	(~C)⊃ASS
If j=i+1:	ASS

3.3 FUNCTION C←A FU B

A function CF←AF FU BF is expected to have already been proved correct with respect to an initial- (I*) and a final- (F*) assertion. Further I*, F* and the header CF←AF FU BF should have been put into ISVAP.

At first in I* and F* the formal parameters CF,AF and BF are substituted by their corresponding actual parameters C,A and B.

Now F* is searched for so called explicit terms of the kind C=f(A,B).

For these terms C is substituted by f(A,B) in ASS using ASSIGN.

Let ASS* be the result of these substitutions. If F* contains explicit terms only, the result is ASS*, otherwise the result is given symbolicly by F* o ASS*, and ISVAP is giving no more help.

,F* o ASS* means that substitutions in ASS* still have to be made by the user, with respect to the remaining implicit terms of F*, using CHANGEASS. (And we hope he can do it!).

Body replacement.

I belive that it is mostly easier to reconstruct a as simplified as possible body of the function FU out of I* and F*. When replacing the functioncall by this body one need no longer propagate through functioncall statements.

Research still has to be made into this subject.

4. Symbol processing in ISVAP

From the previous chapter it became clear that substitution of names and expressions in assertions is an important primitive operation in ISVAP.

4.1 Substitution of names

In ASSIGN and FUNCTION names in assertions sometimes must be renamed temporaly to avoid multiple substitutions afterwards.
This is done by a function STRING←B SUB C which substitutes in STRING the string C for the name B, by compression and expansion, exept in protected parts of STRING.
Examples: 1) rename X by X' (see 3.1)
2) replace formal- by actual parameters (see 3.3)

4.2 Substitution of APL expressions

In ASSIGN and for any explicit term in FUNCTION the left hand side must be substituted by the right hand side in the assertion.
This is done by a function STRING←B ESUB C which substitutes in STRING the APL expression C for the APL expression B. If necessary C is surrounded by parentheses.
Examples: 1) substitute b by b+c (see 1.2)
2) substitute C by f(A,B) (see 3.3)

4.3 Substitution and editing by the user

CHANGEASS allows the user to change assertions or vc's. He can do this by substitutions or editing. When substituting he implicitly uses ESUB. When editing the assertion is printed, followed by a blanc line on which he can indicate with ↑ and ↓ respectively where to introduce or drop symbols.

4.4 Pointers in the program string

The program string is placed in a variable PR.
In order to have a fast reference to individual statements of the program and their constituent parts like LHS and RHS of an assignment, three tables have been introduced.

1) AS, AS[i] contains address and length of LHS and RHS in PR of i-th assignment of the program.
2) TS, TS[j] contains address and length of testcondition in PR of j-th test of the program.
3) FS, FS[k] contains address and length of functionname and each actual parameter in PR of k-th functioncall in the program.

A fourth table TV relates the linenumber of the program to the indices i,j and k of the above tables.

5. The use of APL

5.1 as designlanguage (DL)

Restrictions are necessary rather than extentions.
Restrictions are:

1) No compound statements, like →L,⍴P←A⍴Q←B
2) Only assign A←B or X[V]←B or X[M]←B
   branch →L
   test →(C)/L
   function C←A FU B (all 6 forms).

Extentions: no other than those proposed by other APL programmers.

5.2 as assertionlanguage (AL)

An AL should have at least the same power of expression as the DL.
This means that it should have at least the operators of the DL.
There are two problems (at the moment) when using APL as the AL:

1) APL has no quantors ∀ and ∃.
   In some cases ⍳ can be used to describe the domain:
   (∀i)((i∈⍳n)⊃A[i]=0) can be replaced by ∧/A[⍳n]=0.
   (∃i)((i∈⍳n)∧A[i]=0) can be replaced by ∨/A[⍳n]=0.
   But this is not possible f.i. in:
   (∀i)((i∈⍳n)⊃A[i+0 1]=0) because ∧/A[(⍳n)+0 1]=0.
   This will have to be studied further.
2) Not every mathematical notation can be expressed in APL,
   f.i. greatest common devisor (GCD). A named operator GCD can be introduced..The axioms of this operator have to be cosidered when the vc is proved true.
3) APL has no imlication operator.
   ≤ can be used for it.

5.3 as the descriptionlanguage of ISVAP

There were no difficulties in performing substitutions in the assertionstrings according to the easy addressable parts of the programstring in PR. Only the lack of enquote and dequote operators was felt, f.i. in generating unique names Ti with each time a new "value" i←i+1 for the character i.
Furthermore the programstring should be easily obtainable from the program and vise versa.

TECHNISCHE HOGESCHOOL TWENTE

ONDERAFDELING DER TOEGEPASTE WISKUNDE

5.4 Conclusions about symbol processing aspects of APL in APL program verification with Floyds assertion method.

Realizing that:

1) Substitution, renaming and enquoting and dequoting are almost the only basic symbol processing operations used in program verification by means of assertionpropagation, and
2) Substitution in a string can be easily done by means of compression (deletion of the old entities), followed by expansion (filling in the new entities), and
3) Enquote and Dequote are already under consideration,

it is my feeling that no extentions to APL are necessary.

Theorem proving aspects have not quite well been considered, and quantors still have to be studied. May be they will ask for extentions of the language.

There is however one thing that should have been extended from the very beginning:
namely a very carefull definition of all the operators of APL, especialy unusual situations like empty vectors, because without these axioms it is impossible to prove any vc.

LIMA 0

Patrick MERISSERT-COFFINIERS*

Sommaire

Ce travail a été réalisé dans le cadre du contrat SESORI 73-02I

* CNRS
Laboratoire AL KAOWARIZMI

I Introduction et buts

Les projets LIMA (voir [1] et [2]) portent en général sur des systèmes de mathématiques pures. Celui-ci est un peu en marge, son application principale étant l'obtention de résultats numériques. Nous avons cependant pensé qu'un tel système devait être réalisé, ou son utilité générale.

LIMA 0 traite donc des fonctions réelles d'une variable réelle. Il calcule limites, et développements limités par rapport à une certaine échelle, que nous définirons au § II. Nous verrons dans le dernier § que les projets d'extension comprennent l'introduction de paramètres, puis le passage aux fonctions de plusieurs variables.

I Introduction et buts

Les projets LIMA (voir [1] et [2]) portent en général sur des systèmes de mathématiques pures. Celui-ci est un peu en marge, son application principale étant l'obtention de résultats numériques. Nous avons cependant pensé qu'un tel système devait être réalisé, ou son utilité générale.

LIMA 0 traite donc des fonctions réelles d'une variable réelle. Il calcule limites, et développements limités par rapport à une certaine échelle, que nous définirons au § II. Nous verrons dans le dernier § que les projets d'extension comprennent l'introduction de paramètres, puis le passage aux fonctions de plusieurs variables.

I Introduction et buts

Les projets LIMA (voir [1] et [2]) portent en général sur des systèmes de mathématiques pures. Celui-ci est un peu en marge, son application principale étant l'obtention de résultats numériques. Nous avons cependant pensé qu'un tel système devait être réalisé, ou son utilité générale.

LIMA 0 traite donc des fonctions réelles d'une variable réelle. Il calcule limites, et développements limités par rapport à une certaine échelle, que nous définirons au § II. Nous verrons dans le dernier § que les projets d'extension comprennent l'introduction de paramètres, puis le passage aux fonctions de plusieurs variables.

II Support Mathématique

Nous nous limiterons aux passages à la limite au "voisinage" de l'infini ($+\infty$ plus précisément), tous les autres cas pouvant s'y ramener par un changement de variable. C'est d'ailleurs nous le verrons la technique utilisée par le programme.

L'espace de base dont nous aurons besoin est le suivant :

$\mathcal{C}(\mathbb{R},\infty) = \{(f,a) \mid a \in \mathbb{R}$ et f est une fonction continue sur $[a,+\infty[\}$.

Définissons sur cet espace une relation d'équivalence :

$(f,a)\,\mathcal{R}\,(g,b) \Longleftrightarrow \exists c$; $c \geqslant a$, $c \geqslant b$, et sur $[c,+\infty[$ les fonctions f et g coïncident.

(Cette relation peut être définie sur toutes les fonctions, même non continues).

Soit $\mathcal{F}$ l'espace quotient $\mathcal{C}/\mathcal{R}$, sur lequel se prolongent toutes les opérations usuelles sur les fonctions. En particulier, une classe de $\mathcal{F}$ est dérivable s'il existe un représentant de cette classe (f,a) tel que f soit dérivable sur $[a,+\infty[$. Si cette dérivée est continue, la classe de (f',a) est dans $\mathcal{F}$.

A partir de maintenant, par abus de langage, nous dirons souvent "fonction" au lieu de "classe", parlant de deux "fonctions" égales, ou d'une "fonction" identiquement nulle.

Nous allons maintenant introduire sur $\mathcal{F}$ deux relations d'ordre

$$f \leqslant g \Longleftrightarrow \exists h \underset{a \in \mathbb{R}}{\text{ bornée}} \Big\} \forall x \geqslant a \quad f(x) \leqslant g(x) \times h(x)$$

$$f \ll g \Longleftrightarrow \exists h \underset{a \in \mathbb{R}}{\text{ tendant vers 0}} \Big\} \forall x \geqslant a \quad f(x) \leqslant g(x) \times h(x)$$

Les notations sont celles de Bourbaki. Historiquement les plus célèbres sont celles de Dedekind : O et o qui ont donné son nom au projet qui nous occupe.

équivalence : $f \sim g \Longleftrightarrow (f-g) \ll f$.

On définit d'autres notions utiles :

- f et g sont <u>comparables</u> $\Longleftrightarrow f \ll g$ ou $f \gg g$ ou $f \sim \lambda g$ $(\lambda \neq 0)$
- si g est une fonction positive ayant pour limite 0 ou $+\infty$, f est d'<u>ordre</u> $\rho \Longleftrightarrow \dfrac{\log|f|}{\log g} \to \rho$.

On démontre aisément un lemme très utile :

Lemme A. Si f et g sont deux fonctions positives strictes, f et g sont comparables $\Longleftrightarrow \forall t > 0$ sauf une valeur au plus $f-tg$ garde un signe constant (au sens strict).

Introduisons maintenant la notion fondamentale pour les développements limités :

- On appelle échelle de comparaison tout sous ensemble $\mathcal{E}$ de $\mathcal{F}$ totalement ordonné par $\prec\prec$.

- On dit que $f \in \mathcal{F}$ admet une partie principale par rapport à une échelle de comparaison $\mathcal{E}$, s'il existe $a \in \mathbb{R}$ et $\varphi \in \mathcal{E}$ tels que $f \sim a\varphi$.

- Un développement limité d'une fonction f par rapport à une échelle $\mathcal{E}$ est la donnée de p couples (a_i,φ_i) $(i \in [1,p])$ tels que $a_i \in \mathbb{R}$ $\varphi_i \in \mathcal{E}$ $\varphi_{i+1} \prec\prec \varphi_2$ et $a_{i+1}\,\varphi_{i+1}$ est la partie principale de $f - (a_1\varphi_1 - a_2\varphi_2)$ par rapport à $\mathcal{E}$.

On peut opérer d'une certaine manière, sur les développements limités, les opérations usuelles sur les fonctions : somme produit quotient composition.

- On appelle corps de Hardy un sous-corps $\mathcal{K}$ de $\mathcal{F}$ tel que

$$\forall f \in \mathcal{K} \quad f \text{ est dérivable, et } f' \in \mathcal{K} .$$

Exemple : corps des fractions rationnelles.

Si a et b sont deux fonctions d'un corps de Hardy $\mathcal{K}$ et y une fonction satisfaisant $y' = ay + b$, on peut définir une extension $\mathcal{K}[y]$ du corps $\mathcal{K}$ de la façon suivante :

$\mathcal{K}[y] = \{$fractions rationnelles en y , à coefficient dans $\mathcal{K}\}$.

On démontre sans trop de difficultés que $\mathcal{K}[y]$ est encore un corps de Hardy. En particulier, si $y \in \mathcal{H}$, on peut définir les extensions $\mathcal{H}[e^y]$, $\mathcal{H}[\log|y|]$, $\mathcal{H}[|y|^\alpha]$.

On démontre le lemme suivant

Lemme B. Deux fonctions appartenant à un même corps de Hardy sont comparables d'ordre quelconque ($\forall n$ $f^{(n)}$ et $g^{(n)}$ sont comparables).

Soit $\mathcal{K}_0$ un corps de Hardy, nous pouvons démontrer l'existence d'un sur-corps particulier : il existe un corps de Hardy $\mathcal{K} \supset \mathcal{K}_0$ tel que

$$\forall f \in \mathcal{K} \quad e^f \in \mathcal{K} \text{ et } \log|f| \in \mathcal{K}.$$

La construction de ce corps est la suivante : f est dans $\mathcal{K}$ si $\exists\ \mathcal{K}_1 \ldots \mathcal{K}_n$ corps de Hardy, $u_1 \cdots u_{n-1}$ fonctions telles que :

$$f \in \mathcal{K}_n$$

$$\mathcal{K}_i = \mathcal{K}_{i-1}[u_{i-1}]$$

et $u_i = e^{z_{i-1}}$ ou $\log|z_{i-1}|$, $z_{i-1} \in \mathcal{K}_i$ $z_{i-1} \neq 0$.

Nous pouvons enfin définir le corps des fonctions (H) comme le résultat de cette construction appliquée au corps de Hardy des fractions rationnelles.

Définissons maintenant une échelle de comparaison particulière dans (H). Tout d'abord ℓ_m représentera le logarithme itéré m fois ($\ell_0(x) = x$)

$$\mathcal{E}_0 = \{ \prod_{m=0}^{\infty} (\ell_m(x))^{\alpha_m} | (\alpha_m) \text{ famille nulle presque partout} \}.$$

$\mathcal{E}_0$ est évidemment une échelle de comparaison.

Définissons $\mathcal{E}_n$ par récurrence :

$$\mathcal{E}_n = \{1\} \cup \{ e^{\sum_{k=1}^{p} a_k f_k} | p > 0,\ f_k \in \mathcal{E}_{n-1},\ f_{k-1} \gg f_k \gg 1,\ a_k \neq 0 \}.$$

On voit que $\mathcal{E}_{n-1} \subset \mathcal{E}_n$.

Soit $\mathcal{E} = \bigcup_n \mathcal{E}_n$.

Alors $\mathcal{E}$ est une échelle de comparaison dans (H) vérifiant :

Si $f \in \mathcal{E}$, $g \in \mathcal{E}$, $f \times g \in \mathcal{E}$

Si $f \in \mathcal{E}$, $\mu \in \mathbb{R}$, $f^\mu \in \mathcal{E}$

Si $f \in \mathcal{E}$, $\log f$ est une combinaison linéaire de fonctions de $\mathcal{E}$

Si $f \in \mathcal{E}$ est $f \neq 1$, e^f est équivalente à une fonction de $\mathcal{E}$.

Cette échelle est celle utilisée pour tous les développements limités usuels. Cependant il existe des fonctions (H) qui n'ont pas de partie principale par rapport à $\mathfrak{E}$.

En effet si $f \sim ag \quad g \in \mathfrak{E}$

$$\log f \sim \log(ag) = \log g + \log a$$

et d'après une propriété de $\mathfrak{E}$

$$\log f \sim \Sigma\, a_i g_i \qquad g_i \in \mathfrak{E}$$

c'est-à-dire

$$(\log f - \Sigma\, a_i g_i) \to 0 \ .$$

Or il existe des fonctions ne possédant pas de développement limité à reste tendant vers 0, par exemple :

$$e^{x+\frac{1}{x}} \sim e^x + \frac{e^x}{x} + \frac{e^x}{n!x^n}$$

et la fonction $e^{(e^{x+\frac{1}{x}})}$ n'a pas de partie principale par rapport à $\mathfrak{E}$.

Ceci est une première source de difficultés pour le programme.

Une autre est que l'espace de fonctions autorisées ne sera pas (H) mais l'espace que l'on obtiendrait si, dans la construction de (H), on était parti non pas des fractions rationnelles, mais du corps engendré par les fractions rationnelles et les fonctions $\cos(F(x))$ F étant une fraction rationnelle.

Une fonction comme $\cos(\frac{1}{x})$ n'apporterait pas grand chose de nouveau, car elle même admet un développement limité par rapport à $\mathfrak{E}$; mais il n'en est pas de même de $\cos(x)$!

III Définition du langage

1) Alphabet et lexique

Lettres et chiffres (point décimal compris) sont utilisées, les lettres ne sont pas soulignées. Seules apparaissent : L , LD , ou LG , ayant un sens primitif. Certains assemblages de lettres ont également un sens primitif :

COS SIN EXP LOG +INF -INF . Les signes

- + - × ÷ * ∇ Δ ( )

Dans le cas d'une version comparant un mode définition d'autres signes devraient bien sûr être introduits. Enfin d'autres symboles sont imprimables par le système ´[ ]).

2) Syntaxe

Les chiffres et le point décimal se combinent pour représenter les réels comme en APL . Les noms se composent d'une seule lettre à l'exception des lettres F , G et H qui peuvent être suivies d'un entier explicite. Il y a une notion de type :

"muet" [T.....Z]

"fonction" [F G H] (+ COS SIN EXP LOG)

"constante" : le reste

← représente l'affectation APL

+ , × , ÷ , * sont des opérateurs dyadiques

L , LD , LG , - est dyadique ou monadique.

La syntaxe pour les opérateurs et parenthèses est la syntaxe APL avec l'exception suivante :

si "NOM" est un nom de fonction, alors il doit être suivi d'une parenthèse et ne s'applique qu'à son contenu. Ainsi COS(X)+Y signifie (COS(X))+Y

et non COS(X+Y)

∇ est un opérateur "d'abstraction fonctionnelle" il fonctionne de la façon suivante

[1 lettre de type muet] ∇ une expression LIMA 0 qui n'est pas une fonction.

Exemple : X∇ 3 + COS(2)

Y∇ EXP(Y+1)

Son emploi est légal si l'expression suivant ∇ ne contient pas d'autre lettre de type "muet" que celle précédent ∇. (Une expression qui n'est pas précédée par ∇) ne doit contenir aucune lettre de type "muet").

Le sens de cet assemblage est clair :

si F ← X ∇ EXP(X+1)

F(2) = EXP(3)

F(0) = EXP(1) etc...

Attention ceci n'est pas réellement une fonction APL.

Enfin Δ a un rôle particulier.

Il doit être précédé d'une expression ayant une valeur entière, et suivie d'une fonction, mais son résultat n'est pas une valeur LIMA 0 et ne peut être qu'imprimé immédiatement.

3) Sémantique

L'utilisation des chiffres lettres et ( ) + - × ÷ * — permet de définir des expressions algébriques (et plus avec COS , SIN , EXP, LOG) soit ayant une valeur réelle, soit contenant une lettre de type "muet". Dans ce dernier cas l'application de ∇ permet de définir des fonctions. Nous avons vu dans le § II quelles sont ces fonctions. Elles peuvent à nouveau être combinées à l'aide des opérateurs algébriqu et composées.

L LD et LG représentent les opérateurs limites sur ces fonctions.

Ils peuvent être utilisés comme dyadiques, avec le point limite en 1er argument, ou monadiques, le point limite étant précisé par ailleurs (voir plus loin).

Δ est l'opérateur développement limité. Le premier argument est le nombre de termes désirés (termes non nuls). Le point limite est précisé par ailleurs.

+INF et -INF sont recevables comme valeurs réelles, avec leur signification évidente. Dans le cas d'une limite, le système peut répondre un réel APL , +INF -INF ou UND (pas de limite mais bornée) ou IND (ni limite ni bornes).

L suppose que limite droite et limite gauche sont égales. Si ce n'est pas le cas, l'analyse de la ligne est interrompue, un diagnostic d'erreur est envoyé, et

les valeurs de LD et LG sont imprimées.

Pour Δ le résultat est imprimé comme une fonction, avec la convention que LOG[n] représente un logarithme itéré.

Pourquoi le résultat de Δ n'est-il pas une valeur de LIMA 0 ?

C'est une décision arbitraire. Il me déplaisait de le considérer comme une fonction, bien que ce fût tout à fait possible, et les vecteurs n'existent pas dans LIMA 0 . Mais une modification minimum, indispensable pour toute extension de LIMA 0 et extrèmement aisée à implémenter , serait d'avoir un opérateur monadique ∘ faisant correspondre à une fonction, sa partie principale.

4) Pragmatique

Nous avons des "COMMAND SYSTEMS" analogues à ceux d'APL :)CLEAR)ERASE)FNS)VARS mais surtout un nouveau venu :)SET LIM VAL TO suivi d'une valeur (réelle ou +INF ou -INF). Cette commande a pour effet de fixer le point limite (voir section 3)). Le système répond WAS suivi de la valeur précédente.

IV Organisation du programme et aspect des algorithmes

Comme dans tous les LIMA le centre du programme est un analyseur qui "remonte" la ligne caractère par caractère et appelle les sous-programmes nécessaires : storage de valeurs, appel de valeurs, exécution d'opérateur monadique ou dyadique. Mais en égard au caractère particulier de LIMA deux sous-programmes occupent la moitié du programme RILIM et RIΔL programmes récursifs calculant les limites et les développements limités. Ils sont d'ailleurs étroitement imbriqués, chacun d'eux appelant fréquemment l'autre. Il est d'ailleurs nécessaire d'avoir des repères pour éviter de passer de manière répétée sur les mêmes branches. Etudions de plus près la représentation interne :

Les fonctions sont représentées par un tableau à six colonnes, une ligne par fonction.

La première colonne indique le type de la fonction

0 : fonction constante (la 4e colonne contient alors la valeur)

1 : fonction obtenue par application d'une fonction primitive

alors la 2e colonne indique laquelle : 1 = LOG

2 = EXP

3 = COS

4 = SIN

5 = SIN

la 3e colonne indique la ligne de la fonction à laquelle cette primitive est appliquée

2 : fonction obtenue en composant deux fonctions ou une fonction et une constante avec un opérateur dyadique.

La 2e colonne indique lequel

1 = + 2 = × 3 = ÷ 4 = *

en fait * n'est utilisé que pour le cas f^n

f^g est traduit $e^{g \times \log f}$

(f-g est traduit $f+(-g)$)

des doubles (3e-4e colonne) et (5e-6e colonne) sont utilisés pour les arguments de cet opérateur, soit : (ligne de la fonction, 0) ou (0, valeur de la constante).

Quand un calcul asymptotique est demandé, un autre tableau est engendré parallèlement au tableau des fonctions. Il contient bien sûr des pointeurs menant à l'information calculée mais aussi des indications sur la quantité d'information déjà calculée.

On peut tout d'abord en voir une application immédiate :

$$G \leftarrow \underbrace{(\ldots F \ldots)}_{1} + \underbrace{(\ldots F \ldots)}_{2}$$

Dans un calcul sur G , un calcul sur F sera effectué lors du traitement de la parenthèse 1, il ne sera pas réeffectué lors du traitement de la parenthèse 2. Un gain beaucoup plus important peut intervenir dans le cas de formes indéterminées

$$F \leftarrow F1 - F2$$

Le système reçoit maintenant l'ordre de calculer p termes du développement limité de F . Il suppose à priori qu'il lui faut p termes pour F1 et F2 , et les calcule. A ce moment par simplifications F1 - F2 donne moins de p termes. Il faut donc revenir en arrière et calculer des termes supplémentaires de F1 et F2 (à priori autant qu'il manque de termes à F1 - F2). A ce moment là, il faut éviter de calculer <u>tout</u> le développement de F1 et F2 . L'intérêt est d'autant plus grand que F1 est complexe. Si F1 = COS l'intérêt est presque nul. Si F1 est obtenu par composition de deux fonctions le gain peut être très grand.

(<u>Remarque</u> : le problème de la simplification étant vraiment très éloigné des intérêts du projet, aucun gros effort n'a été fait en ce sens. Il est donc possible de trouver un cas où la procédure précédente ne converge pas, c'est-à-dire, où on n'obtienne jamais p termes pour F1 - F2 , c'est à dire où le nombre de termes calculés soit stationnaire. Alors le programme engendre la fonction F1 - F2-termes déjà calculés, et teste empiriquement si elle est identiquement nulle, en la calculant pour quelques valeurs aléatoires).

Un autre cas où il faut peut être recalculer des termes est le cas

$$F \leftarrow EXP(G) \quad \text{et} \quad G \rightarrow +INF$$

Le cas limite étant celui vu au § II où F n'admet pas de développement limité par rapport à $\mathfrak{G}$, et où le programme doit répondre par F , aveu d'impuissance.

V Une session

Voici le listing commenti d'une session réalisée sur IBM 360 (Pise Février 1974).

Les questions les plus difficiles ont reçu une réponse en 30 secondes de temps réel, la plupart des autres en quelques secondes.

Tout d'abord, quelques exemples de développements limités de fonctions simples : exponentielle, cosinus, et sinus hyperbolique.

```
      TEST
)SET LIM VAL TO 0
    WAS +INF
F←X∇EXP(X)
4 Δ F
    1+X+0.5×(X*2)+0.166666×(X*3)+0.041666×(X*4)
F1←X∇COS(X)
2 Δ F1
    1+¯0.5×(X*2)+0.041666×(X*4)
G←Z∇(EXP(Z)-EXP(-Z))÷2
3 Δ G
    X+0.166666×(X*3)+0.008333×(X*5)
)FNS
    F F1 G
)CLEAR
```

Développement limité de fonction composée :

```
F←X∇(X+EXP(X))
3 Δ F
    1+2×X+0.5×(X*2)+0.166666×(X*3)
F←X∇LOG(F(X))
3 Δ F
    2×X+¯1.5×(X*2)+1.833333×(X*3)
```

```
F←XV(X-SIN(X))÷X*3
L F
     0.166666
```

Deuxième exemple de forme indéterminée : 1 * ∞

```
)SET LIM VAL TO +INF
     WAS 0
G←XV(1+1÷X)*X
A←LOG(+INF L G)
A
     0.999999
```

Troisième exemple de forme indéterminée : ∞ - ∞

```
F←XV(X×(1+X*2)*1÷2)-X*2
L F
     0.5
```

Fonctions mêlant puissances de X, exponentielle, et logarishmes itérés :

```
G←XVLOG(X)+X+LOG(LOG(X))*2
3 Δ G
     X+(LOG(X))+(LOG[2](X)*2)
H←XVEXP(G(X))
1 Δ H
     EXP(X+(LOG[2](X)*2))×X
```

Exemples de limites à droite et limites à gauche :

```
)SET LIM VAL TO 0
     WAS +INF
F←X∇1÷X
L F
     LC: -INF    LD: +INF
     DOMAIN ERROR
     LF
     ∧
LG F
     -INF
F←X∇EXP(1÷X)
L F
     LG: 0    LD: +INF
     DOMAIN ERROR
     LF
     ∧
G←X∇EXP(-1÷X*2)
L G
     0
```

Exemples de fonctions n'ayant pas de limite :

```
)SET LIM VAL TO +INF
     WAS 0
F←X∇COS(X)
L F
     UND
G←X∇X×COS(X)
L G
     IND
```

Pour finir, un exemple de développement limité au voisinage d'une valeur finie non nulle.

```
)SET LIM VAL TO 1
     WAS +INF
F←X∇LOG(X)
2 Δ F
     (X-1)+¯0.5×((X-1)*2)
)OFF
```

VI Conclusions

1) Remarques sur APL comme support

Certaines faiblesses d'APL ont considérablement augmenté l'aspect pénible de l'implémentation : l'absence de tableaux (de tableaux). Une fonction est un vecteur, un développement limité est un vecteur de fonctions, on s'intéresse à un tableau de développements limités... Il a souvent été nécessaire de définir trois tableaux là où un aurait dû suffire ; et de compliquer la programmation en conséquence.

D'autre part une forme de recursivité parait assez utile :

Prenons un exemple plus simple que le calcul asymptotique : la suite $u_{n+2} = u_{n+1} + u_n$. Elle correspond en APL au programme suivant :

```
∇R ← F N
  R ← 1
→ (N < 2)/0
R ← (F N-2) + F N-1 ∇
```

Pour calculer u_2 ce programme calcule u_1 et u_o.
Pour calculer u_3 ce programme calcule 2 fois u_1 et 1 fois u_o.
Pour calculer u_4 ce programme calcule 3 fois u_1 et 2 fois u_o (et en cours de calcul 2 fois u_2).
Dans le calcul de u_n, le programme calcule 2 fois u_{n-2}, ... u_{n-2} fois u_2 ... u_{n-1} fois u_1.

Il serait utile d'engendrer, au début du calcul un tableau à 2 colonnes, avec T[N;1] indiquant si F N a déjà été calculé, et si oui, son résultat dans T[N;2]. Bien sûr ceci n'est pas le cas dans tous les programmes récursifs que l'on veut écrire ; il faudrait avoir cette possibilité en option.

2) Amélioration des algorithmes

On m'a suggéré qu'il devait être possible, par exemple dans le cas des formes indéterminées, de prévoir avec peu ou pas de calculs, le nombre de termes à calculer pour les fonctions intermédiaires. Toute suggestion concrète et efficace sera exa-

minée avec intérêt et reconnaissance !

3) Extensions possibles

Tout d'abord, précisons que LIMA 0 n'étant qu'un épisode dans un projet général, ces extensions ne seront probablement pas implementées.

La première idée est d'introduire des paramètres, c'est-à-dire des constantes à valeur non précisée. Les réponses du système à un ordre de passage aux limites, serait le cas échéant un arbre.

Puis dans une fonction avec paramètres, on pourrait échanger le rôle de la variable avec celui d'un des paramètres, ce qui serait le premier pas vers le traitement des fonctions à plusieurs variables.

Une extension différente et facilement réalisable, serait l'introduction d'un mode définition.

Admettons l'opérateur o mentionné en III 3).

Voici deux exemples de problèmes où un mode définition serait agréable :

a) Tout d'abord l'équation $Y\ LOG(Y) = (X)$ qui se transforme en

$LOG(Y) + LOG(LOG(Y)) = LOG(X)$ ou $U + LOG\ U = LOG\ X$ avec $U = LOG\ Y$ d'où

$U \sim LOG\ X$.

On peut calculer le terme suivant : $U = LOG\ X + u_1 \qquad u_1 + LOG(LOG\ X + u_1) = 0$

$u_1 \sim -LOG(LOG(X))$. Au stade suivant cela donne

$u_2 = LOG(LOG(X)) - LOG(LOG\ X - LOG(LOG\ X) + u_2)$

$u_2 = -LOG(1 - \frac{LOG(LOG(X))}{LOG(X)} + \frac{u_2}{LOG(Y)})$

$u_2 \sim \frac{LOG(LOG(X))}{LOG(X)}$.

Ce calcul correspond à la boucle

```
     I ← 1
F1 ← F2 ← X ∇− LOG(LOG(X))
→ (N=I)/FIN
INIT : I ← I+1
F1 ← ∘ − LOG(LOG+F2) + F2
```

F2 ← F2 + F1

→ INIT

b) Considérons l'équation $x = \operatorname{tg} x$ et appelons u_n la solution comprise entre $\frac{(2n-1)}{2}\pi$ et $\frac{(2n+1)}{2}\pi$. Nous cherchons un développement de u_n par rapport à n. Visiblement $u_n \sim \frac{(2n+1)}{2}\pi$

$$u(n) = n\pi + \frac{\pi}{2} - u_1(n)$$

avec

$$\operatorname{tg}(n\pi + \frac{\pi}{2} - u_1(n)) = n\pi + \frac{\pi}{2} - u_1(n)$$

$$\operatorname{tg}(u_1(n)) = \frac{1}{n\pi + \frac{\pi}{2} - u_1(n)}$$

$$u_1(n) \sim \frac{1}{\pi n}$$

ce qui donne lieu à une boucle comme la précédente, en admettant qu'on ait déjà résolu le problème d'Arctg.

Bibliographie

[1] P. BRAFFORT. Declarations et initialisations, Note ECSTASM n° 1 .

[2] D. FELDMANN et P. MERRISSERT-COFFINIERES. LIMA ω , Note ECSTASM n° 3 .

[3] BOURBAKI. Fonctions réelles de variables réelles.

APS : A CONVERSATIONAL ALGORITHMIC PROGRAMMING SYSTEM

G. Aguzzi, R. Pinzani, R. Sprugnoli

1. Abstract syntax and semantics of λ-calculus
2. Concrete representation of the syntax and semantics of the SECD machine.
3. A sketch of the implementation of APS in APL

* Consiglio Nazionale delle Recerche ITALY

1. Abstract syntax and semantics of λ-calculus.

Following Landin [1], the semantics of λ-calculus can be given in an interpretative way by means of the concept of "state transformation".

The semantics of a λ-expression is so defined by showing the successive transformations of a particular object called "state". A state is a composite object whose components are relevant in order to exactly define the meaning of a λ-expression. Such components are: a control part whose content is the λ-expression whose meaning is to be defined, a stack part which is used in order to store evaluated argument of expressions, an environment part which contains symbols and associated values to be used in the computation of the expression present in the control part, a dump part which contains a copy of the preceding state, i.e. the state saved whenever a new level of expression evaluation is entered.

A state transformation is performed by a particular function whose argument is a state and whose output is still a state. Such a function, called transform precisely defined by Landin, operates these state transformations starting from an initial state x_o which has his control part containing the λ-expression to be evaluated and all other components empty,(+) thus giving a new state x_1 and recursively from x_i x_{i+1} until a final state x_f is reached. It is just this finite set of states which gives the semantics of the particular λ-expression present in the state x_o. This function may be also called an "abstract machine" performing the above mentioned transformations; Landin called

---

(+) with the exception of the environment part which may contain the variable-value pairs, for the free variables of the λ-expression in the control part.

his machine the "SECD machine" because it operates on a Stack, an Environment, a Control and a Dump part.

1.1 - Syntax

In order to precisely define this function we need a syntactic specification of the involved objects. Such a specification can be also useful in the definition of the function itself. In this section we give these specifications in a complete abstract way, also giving the function transform operating on the abstract objects defined by the abstract syntax. In the following sections it will be also shown how concrete representations can be derived from the abstract notation. For the moment, however, we should like to show the "intuitive" abstract syntax of the objects involved in the specification of the SECD machine. We use here a pseudo-context-free notation for the definition of objects in the sense that the symbol "::=" is replaced by the symbol "=" and its meaning is "is composed by", the symbol " " (blank) is used in order to separate the names of objects possibly occurring as components in the right member of each production, and the symbol "|" is used as the usual alternative "or". In this way a production like:

machine = stack env control dump

reads as follows:
the object machine is composed by a stack part and by an env part and by a control part and by a dump part (we use here the term machine in the same sense of the term state we used above) and, analogously, a production like:

env = Ω | envelement env

reads as follows:
the object env is composed by the Ω (null) object or by an envelement part and by an env part.

We can give now our intuitive abstract syntax of the objects involved in the SECD machine specification:

(IAS1) machine = stack env control dump
(IAS2) stack = Ω | stackelement stack
(IAS3) stackelement = closure | expression
(IAS4) env = Ω | envelement env
(IAS5) envelement = var value
(IAS6) control = Ω | contrelement control
(IAS7) contrelement = expression | ap
(IAS8) dump = Ω | machine
(IAS9) closure = env var expression
(IAS10) expression = var | λexpr | combination
(IAS11) λexpr = boundvar body
(IAS12) boundvar = var
(IAS13) combination = rator rand
(IAS14) rator = expression
(IAS15) rand = expression
(IAS16) value = closure | expression
(IAS17) body = expression

in this definition Ω stands for the null object
ap for an elementary object not ∈ var
var for a set of elementary objects.

From IAS10 to IAS17 we can see that an expression is an elementary object var or a composite object λexpr or a composite object combination. A λexpr is an object in which we may distinguish a boundvar part (i.e. an object which is, on its turn from IAS12, a var) and a body part; a body is still an expression (see IAS17).

A combination ia an object composed by a rator and a rand which are both expressions.

This kind of definition is what Landin calls a "structure definition" of the objects we are interested in. No committements are made about any kind of concrete representation of such objects. Following rules IAS1 to IAS17 it is a quite natural job to derive a formal definition of these objects in terms of Vienna Definition Language (VDL) [2], i.e. defining each composite object by means of a predicate which is satisfied only by those objects constructed in the way illustrated in the definition of the predicate itself by means of other (possibly composite) objects, particular selectors and other predicates.

We give here a rough explanation of this definition method in order to show the complete abstract syntax of our objects which may be called from now on "abstract trees".

We recall here that we are dealing now with two data object classes, namely the class EO of elementary objects (i.e. atomic objects or terminal nodes of trees) and the class CO of composite objects which may be built up from elementary ones by construction operators (see below); composite objects have components each of which may be selected by one and only one selector choosen from a class S of selectors.

From IAS2 we can derive the following definition:

$$\text{is-stack} = \text{is-empty} \vee (\langle \text{s-stackelement: is-stackelement} \rangle, \langle \text{s-stack: is-stack} \rangle)$$

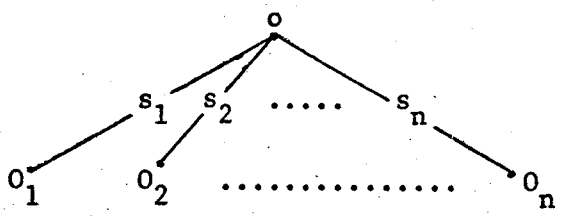
that is the predicate "is-stack" (derived from the name stack with the prefix "is") is satisfied (this is the meaning of the sign "=")

by the null object Ω , which satisfies the is-empty predicate, or is satisfied by those objects which have just two components labelled s-stackelement and s-stack and satisfying the is-stack-element (see below) and is-stack predicate respectively.
The second alternative used in this definition parallels the way in which composite objects may be constructed using the construction operator μ_o of VDL. The operator μ_o: $(Sx(EO \cup CO))^n \rightarrow CO$, maps an n-tuple of couples of type (s_i, O_i) where $s_i \in S$ (selector) and $O_i \in (EO \cup CO)$ into a composite object $\sigma \in CO$, with the condition that for every i and j with $i \neq j$ and $i,j \leq n$ $s_i \neq s_j$.
Couples of the type (s_i, O_i) will be denoted by $\langle s_i : O_i \rangle$.
In this way the application of μ_o to the following n-tuple of couples:

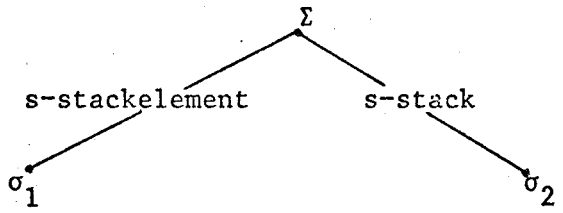
$$\langle s_1 : O_1 \rangle , \langle s_2 : O_2 \rangle , \ldots , \langle s_n : O_n \rangle \quad (\text{with } s_i \neq s_j \text{ for } i \neq j \text{ and } i,j \leq n)$$

builds up the composite object whose immediate components are labelled $s_1, s_2, \ldots, s_n$ and select the objects $O_1, O_2, \ldots, O_n$ respectively.
Graphically we have:

$$\sigma = \mu_o(\langle s_1 : O_1 \rangle, \ldots, \langle s_n : O_n \rangle) \equiv$$



Coming back to the is-stack predicate definition, given an object σ_1 satisfying the is-stackelement predicate and an object σ_2 satisfying the is-stack predicate, an object Σ satisfying the is-stack predicate is the null object or the object graphically represented by;



We are now in a position to give the complete abstract syntax derived from IAS1 - IAS17 by means of predicates, selectors and the underlying concept of the μ_o constructor operator. Selectors will be freely used in each rule without previously defining a set S of selectors; this set S results in this way a posteriori defined as the union of all the used selectors.

(AS1) is-machine = (<s-stack:is-stack>,<s-env:is-env>, <s-control:is-control>,<s-dump:is-dump>)

(AS2) is-stack = is-empty ∨ (<s-stackelement; is-stackelement>, <s-stack: is-stack>)

(AS3) is-stackelement = is-closure ∨ is-expression

(AS4) is-env = is-empty ∨ (<s-envelement: is-envelement>, <s-env· is-env>)

(AS5) is-envelement = (<s-var: is-var>, <s-value: is-value>)

(AS6) is-control = (⟨s-contrelement: is-contrelement⟩, ⟨s-control: is-control⟩) ∨ is-empty

(AS7) is-contrelement = is-expression ∨ is-ap

(AS8) is-dump = is-machine ∨ is-empty

(AS9) is-closure = (⟨s-env: is-env⟩, ⟨s-var: is-var⟩, ⟨s-expression: is-expression⟩)

(AS10) is-expression = is-var ∨ is-λexpr ∨ is-combination

(AS11) is-λexpr = (⟨s-boundvar: is-var⟩, ⟨s-body: is-expression⟩)

(AS12) is-combination = (⟨s-rator: is-expression⟩, ⟨s-rand: is-expression⟩)

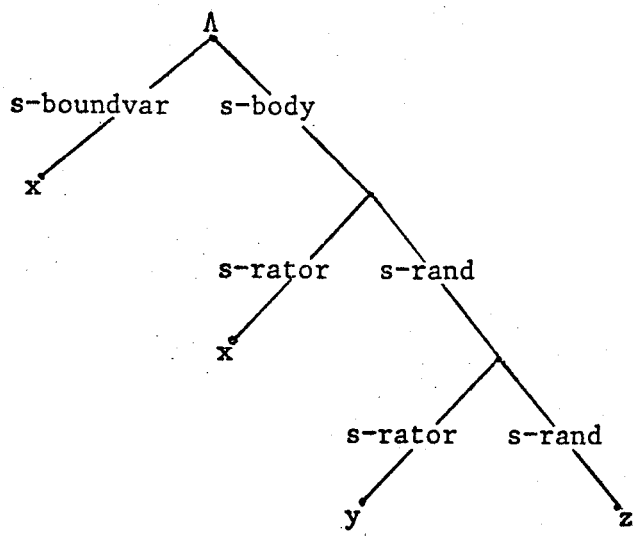
(AS13) is-value = is-closure ∨ is-expression

(AS14) is$\hat{}$var = {a set of elementary objects}

(AS15) is$\hat{}$ap = an elementary object $\notin$ is$\hat{}$var

note that is$\hat{}$var and is$\hat{}$ap are the sets defined by the predicates is-var and is-ap respectively.

Assuming, for example, is$\hat{}$var = {x,y,z,f,g,h} an object Λ, satisfying the is-λexpr predicate may be so represented:



Selectors may be thought of as operators which whenever applied to objects belonging to OB = EO ∪ CO, give as result the object ∈ OB which is attached to the branch labelled with the selector itself (if it exists) or the null object otherwise. So we have, for example,

s-boundvar (Λ) = x
s-rator (s-body (Λ)) = x
s-rator (s-rand (s-body (Λ))) = y
s-stack (Λ) = Ω

1.2 - Semantics

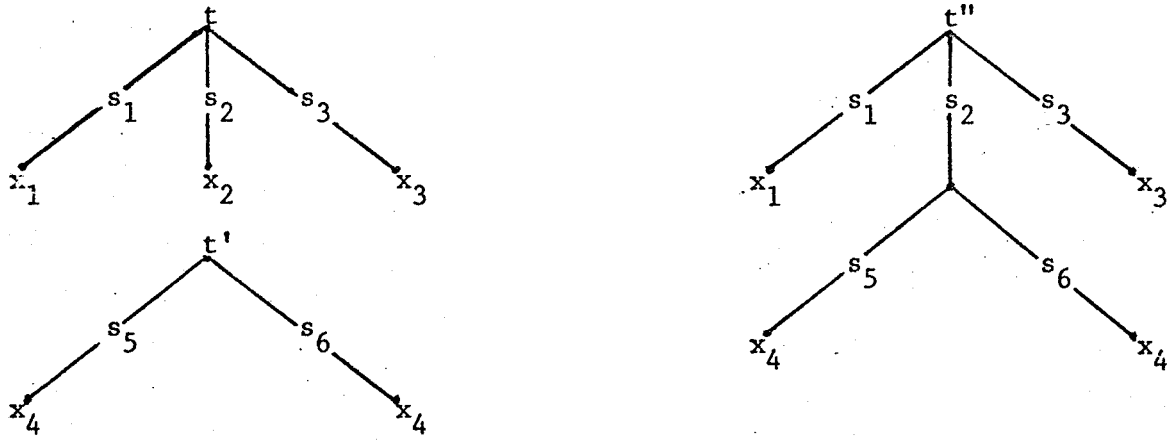
In order to give the definition of the transform function, we need some more formalism and conventions. In the sequel the so called generalized assignment operator μ = OBxSxOB → CO ∪ Ω , will be used.
This operator is introduced in order to be able to update and manipulate objects, and is a generalisation of the operator μ_o [2]. Given an object t, a selector s and an object t', performing

$$\mu\ (t;\ \langle s:t' \rangle\ )$$

we have as result the object t' assigned to the s-component of t if t has an s-component, or the addition of a new s-component t' to t if t does not have an s-component; or the deletion of s(t) if t' = Ω .
Thus, for example, let $t = \mu_o\ (\langle s_1:x_1\rangle, \langle s_2: x_2\rangle , \langle s_3: x_3\rangle\ )$ performing

$$\mu\ (t;\ \langle s_2: \mu_o\ (\langle s_5: x_4\rangle, \langle s_6: x_4\rangle\ )\rangle) = t''$$

we have



i.e. we have the assignment of a new value to the s_2-component.

A function operating on objects $\in OB$ may be defined by means of a sort of conditional expression language, using the predicates and selectors yet introduced in the syntax definition and the logic operators.

The usual keywords if and then are not used here and only the predicate following the if, i.e. the "if clause", is written; the keyword then is replaced by an arrow " → ". The only primitive in this language is the substitution of objects by other objects into a predefined object (i.e. the object on which the function operates). A substitution is indicated by writing down as l.h.m. the selector giving the object to be substituted, then an assignment arrow " ← " and, as right hand member, the actual object which replaces the old one.

Thus, for example, operating on the object m satisfying the is--machine predicate an alternative may be so represented in our

conditional expression language

is-empty (s-control (m)) →

s-stack ← μ_o(<s-stackelement: s-stackelement (s-stack (m))>,
<s-stack: s-stack (s-dump (m))>)

is-var (s-contrelement (s-control (m)) → .....

which means that if the predicate is-empty applied to the control part of the machine m is true then substitute the s-stack-component of m with the object desçribed in the right hand side of the substitution rule, else if is-var (s-contrelement (s-control (m)) is true then etc. Obviously all these transformations could be described using the μ operator only, but we prefer to give here a less formal description of our transform function.
So we can define our transform algorithm, using this language.
This algorithm operates on a state or machine m satisfying the AS1 predicate.
For shortness purposes we put

S = s-stack (m)
E = s-env (m)
C = s-control (m)
D = s-dump (m)
P = is-closure (s-stackelement (S))
S' = s-stack (s-dump (m))
E' = s-env (s-dump (m))
C' = s-control (s-dump (m))
D' = s-dump(s-dump (m))

We assume as primitive the following function

val: (is^env x is^var) → is^value

which for every variable gives the value associated to it in the environment part of the machine (see IAS5 and AS5).
Finally we can give

transform (m)

t1) is-empty (C) →

s-stack ← μ_o(<s-stackelement: s-stackelement (S)>,<s-stack: S'>)
s-env ← E'
s-control ← C'
s-dump ← D'

t2) is-var (s-contrelement (C)) →

s-stack ← μ_o(<s-stackelement: val (E, s-contrelement (C))> ,
<s-stack: S>)
s-control ← s-control(C)

t3) is-λexpr (s-controlement(C)) →

s-stack ← μ_o(<s-stackelement: μ_o(<s-env: E>,<s-var: s-boundvar
(s-contrelement (C))>,<s-expression: s-body
(s-contrelement (C))>)>,<s-stack: S>)
s-control ← s-control (C)

t4) is-ap (s-contrelement (C))$\wedge$P →

s-stack ← Ω
s-env ← μ_o(<s-envelement: μ_o(<s-var: s-var (s-stackelement (S))>,
<s-value: s-stackelement (s-stack (S))>)> ,
<s-env: s-env (s-stackelement (S))>)
s-control ← μ_o(<s-contrelement: s-expression (s-stackelement (S))>,
s-dump ← μ_o(<s-stack: s-stack (s-stack (S))>, <s-control: Ω>)
<s-env: E> , < s-control: s-control (C)> ,
<s-dump: D>)

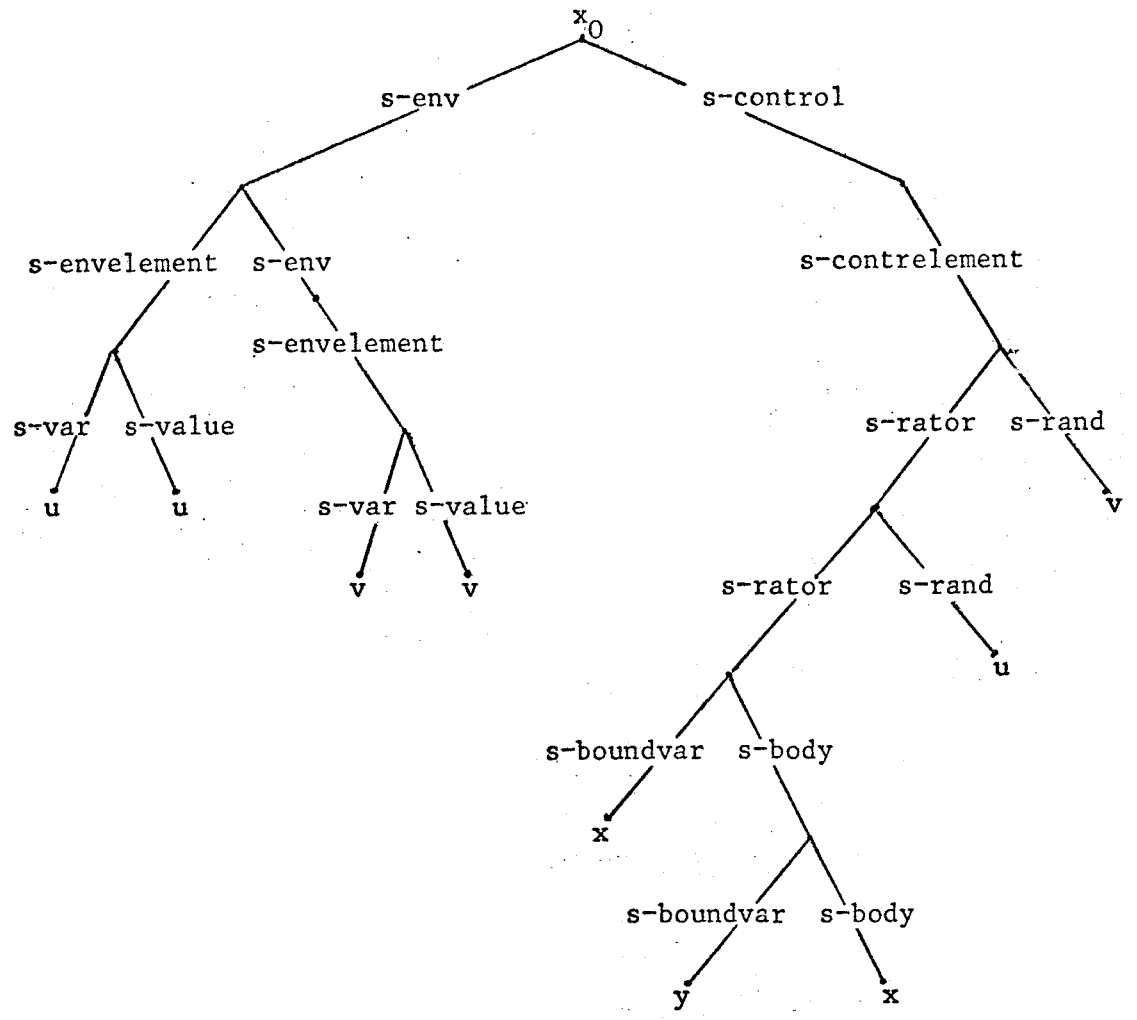
t5) is-ap (s-contrelement (C)) $\wedge \sim$ P $\rightarrow$

s-stack $\leftarrow \mu_0$(<s-stackelement: μ_0(<s-rator: s-stackelement (S)>,
<s-rand: s-stackelement (s-stack (S))>)> ,
<s-stack: s-stack (s-stack (S))>)
s-control $\leftarrow$ s-control (C)

t6) is-combination (s-contrelement (C)) $\rightarrow$

s-control $\leftarrow \mu_0$(<s-contrelement: s-rand (s-contrelement (C))> ,
<s-control: μ_0(<s-contrelement: s-rator
(s-contrelement (C))>,<s-control:
μ_0(<s-contrelement: is-ap>,<s-control:
s-control (C)>)>)>)

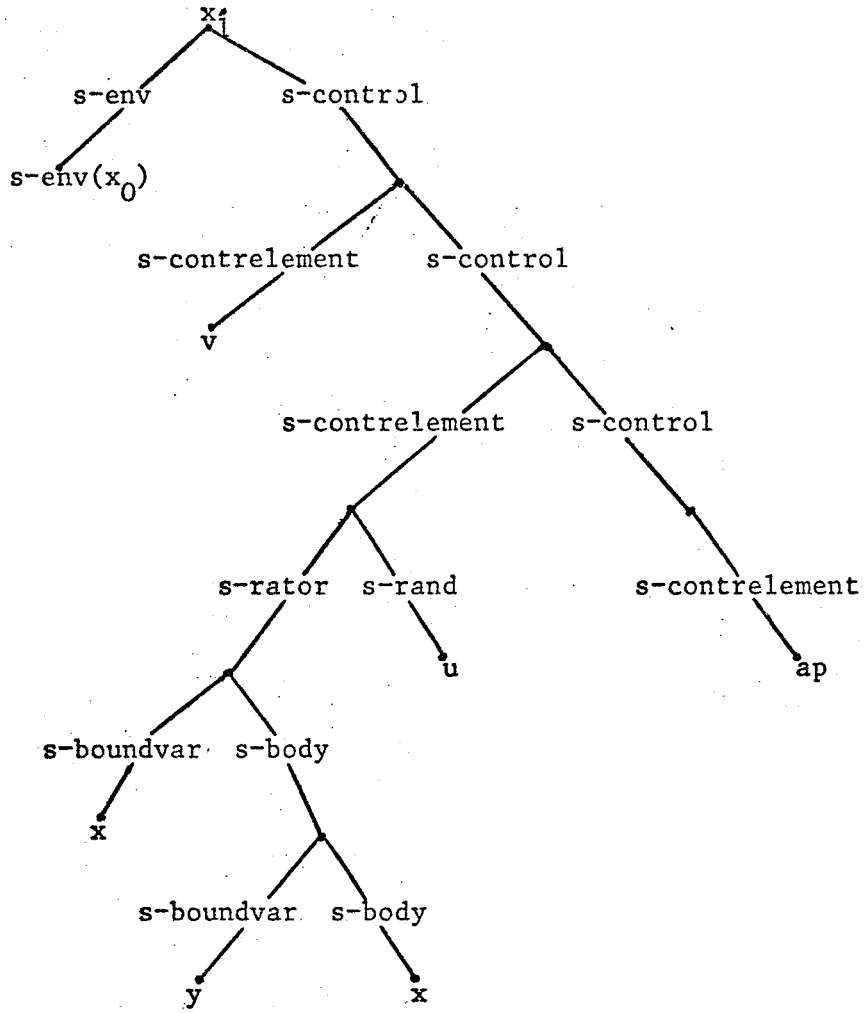
This function accomplishes the above mentioned task in the sense that for every state x_i gives as result a new state x_{i+1} whose components may be a modification with respect to those of x_i.
Let us illustrate some of transformations with a simple example.
Example: let x_o be the following object (note that the initial state contains the expression to be evaluated as its control part and a certain environment part which provides a "value" for each variable which is free, i.e. never selected by an s-boundvar selector, in the expression):



Note that x,y,u,v $\in$ is^var.

Applying the transform function we see that only predicate t6 is satisfied; it tests for a combination; this is, in fact, the structure of our expression. The effect of the application of t6 is the transformation of the control part of x_0 in the following way: put as contrelement part of the control part of x_1 the rand part of the original expression, then instal a control part which is composed by a contrelement part consisting of the rator part of the expression and a control part which is, on its turn, composed by a contrelement part yelding an "ap" symbol (supposing this symbol satisfies the is-ap predicate) and a control part consisting in the control part of C which is, in

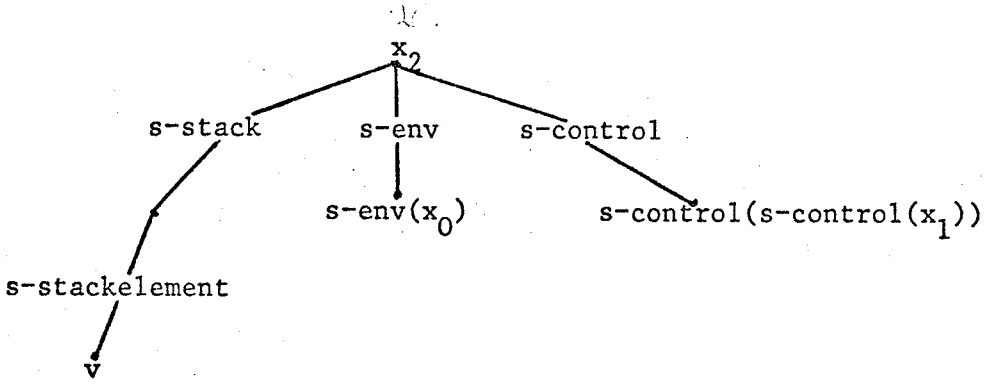
this case, empty thus obtaining



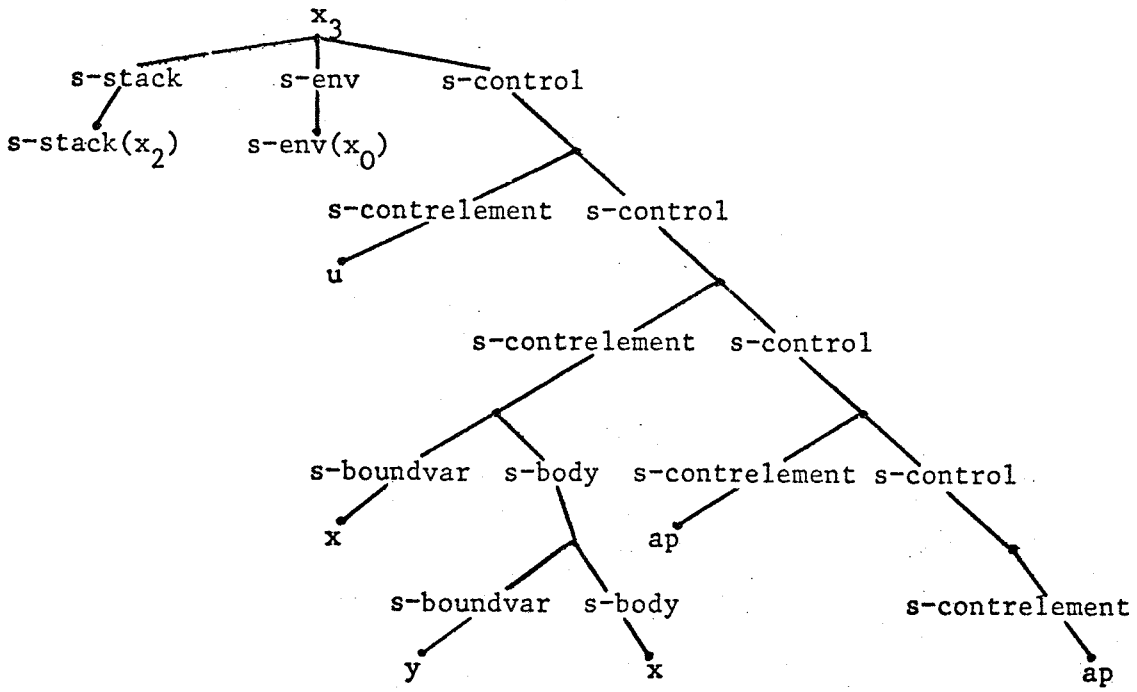
where s-env (x_o) means the s-env component of x_o since this component is unchanged.

The strategy of evaluation of combinations consists first in the evaluation of its (ope)-rand part and then in the evaluation of the (ope)-rator part. This is of course one of the possible strategies, but we do not discuss here the problem, which has

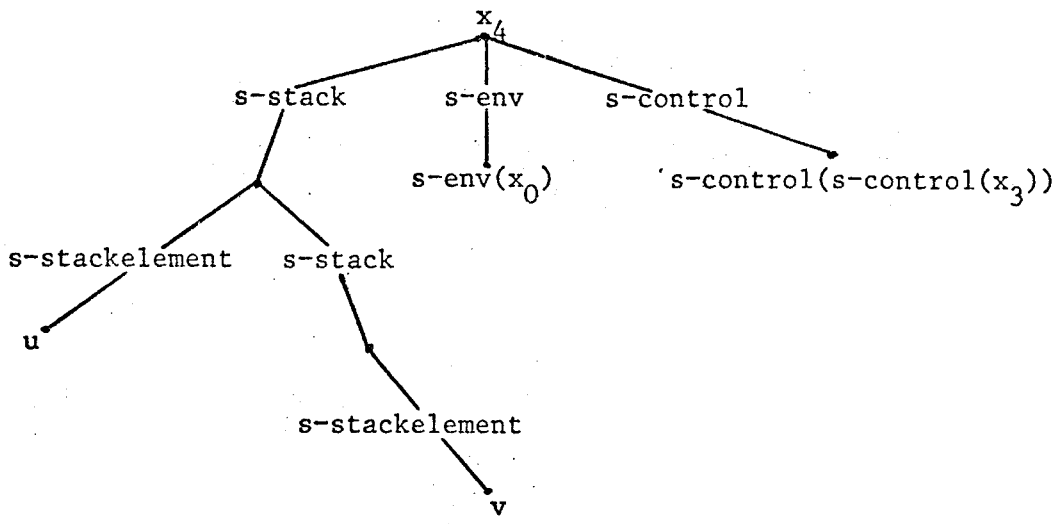
been carefully studied (see e.g. [7]). We have so obtained a state x_1, to which we apply again our transform function; we see that the t1 predicate is satisfied thus obtaining the transformation of the stack part of x_1 and the deletion of the contrelement part of C in x_1. The resulting state x_2 is the following:



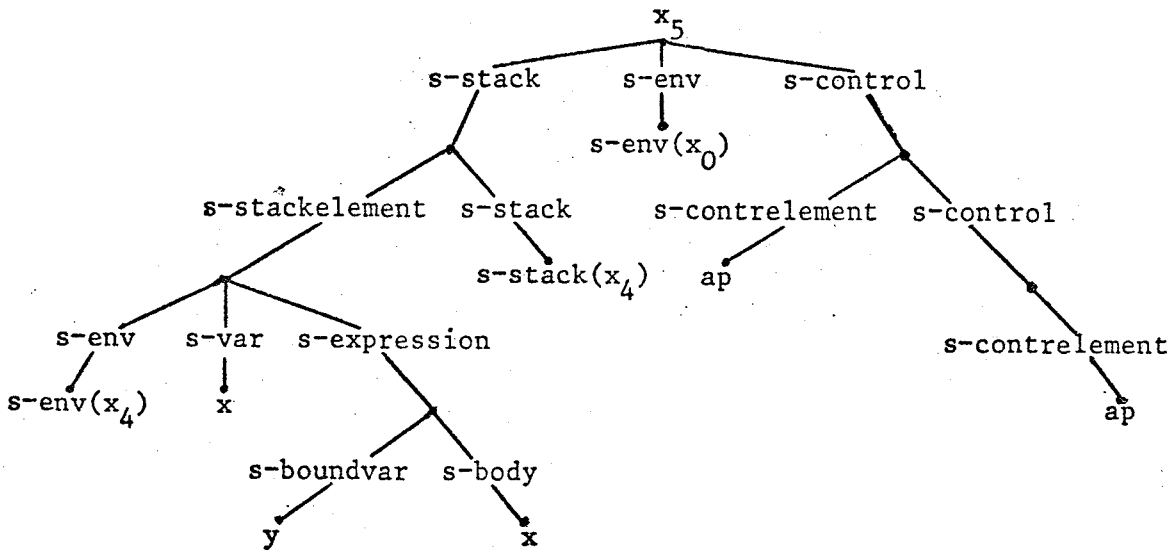
The evaluated rand part is pushed down as stackelement of the stack part of x_2. We have in fact val (E,v) = v, i.e. in the environment the value v is associated to the variable v.
Now the alternative t6 becomes applicable, since we have a contrelement part which is a combination, thus obtaining:



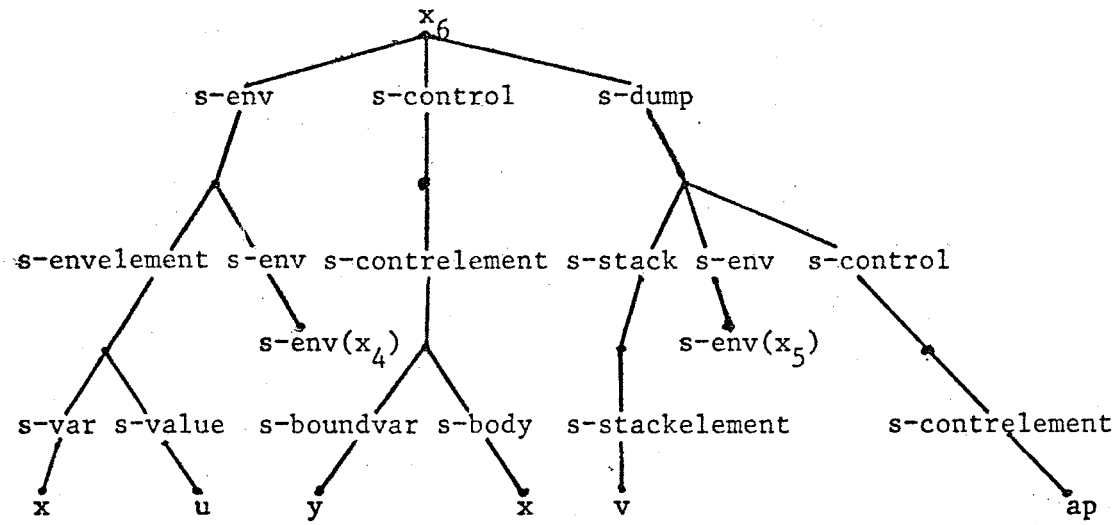
As we can see another ap symbol appears as last element of the control part.
Alternative t2 is applicable, and recalling that val (E,u) = u we obtain:



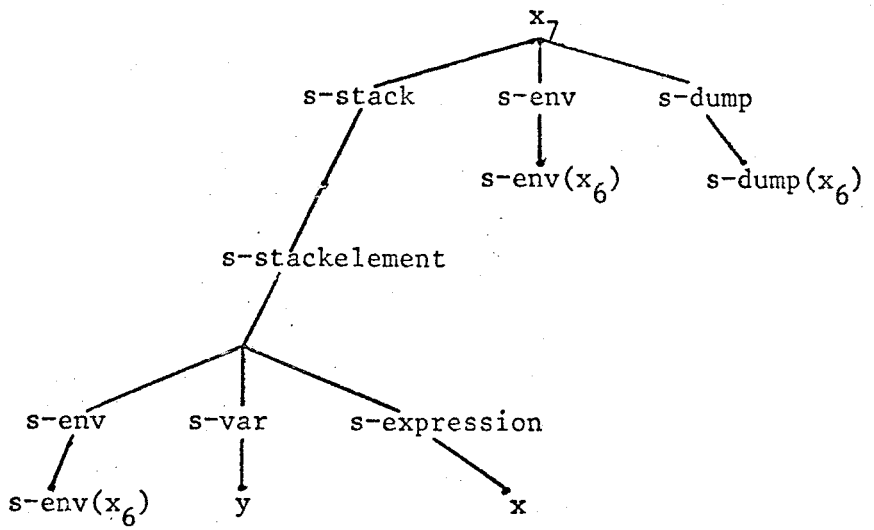
The first applicable alternative is now t3, which tells us that the contrelement part is a λ-expr structure. In the stack part is so pushed the object called "closure", built up by means of E, the boundvar part of the λ-expr and the body of the λ-expr itself. The resulting x_5 state is:



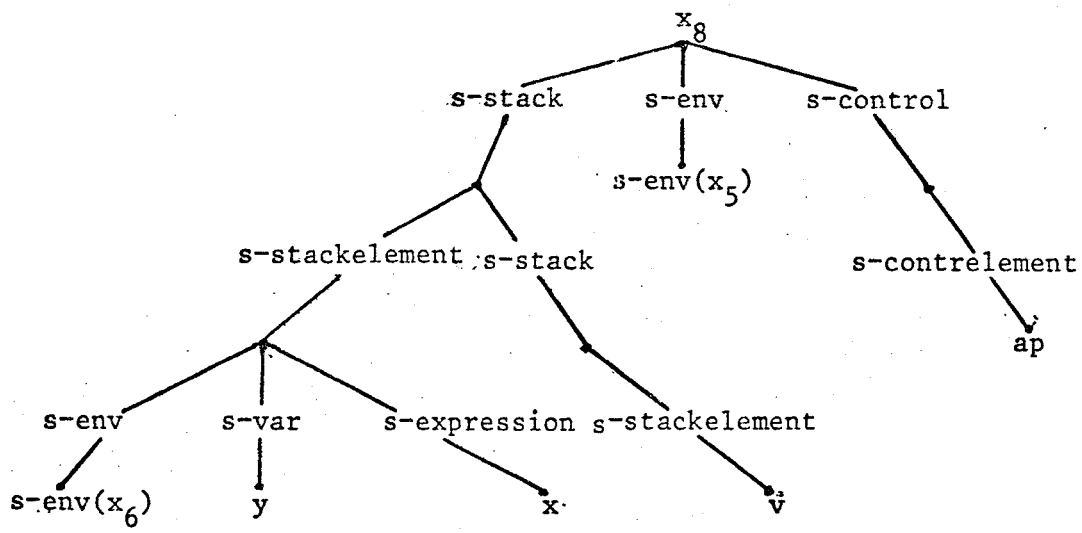
At this point alternative t4 is satisfied. We have in fact an ap symbol.as contrelement part of C and a closure (see AS9) as stackelement part of S. This alternative produces the deletion of the stack, the installation of the environment part of the closure, updated by inserting the new variable-value couple formed by the variable part of the closure and the stackelement part of the actual stack part of S, the insertion of the expression part of the closure as new contrelement of the control and finally the storing of the state x_5 in the dump part, noting that the stack part is stack (stack (S)). This means that a new level of evaluation is entered. We thus obtain:



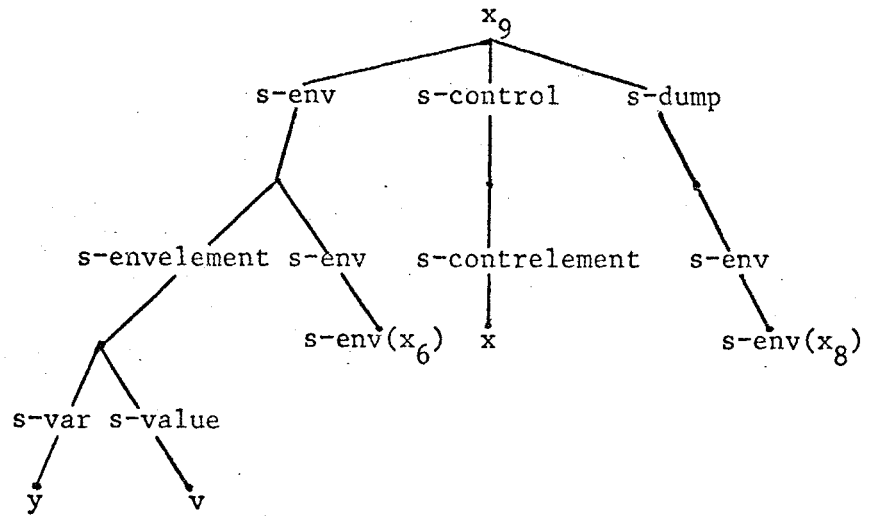
Alternative t3 is now satisfied thus obtaining the insertion as stackelement of the stack the new closure, and an empty control part; state x_7 is:



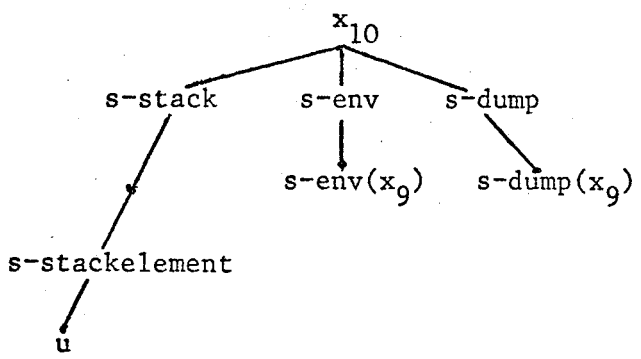
Since alternative t1 is satisfied, the various components of the dump are popped up; however, the stackelement part of the stack in x_7 is still the stackelement part of the stack in the new state.



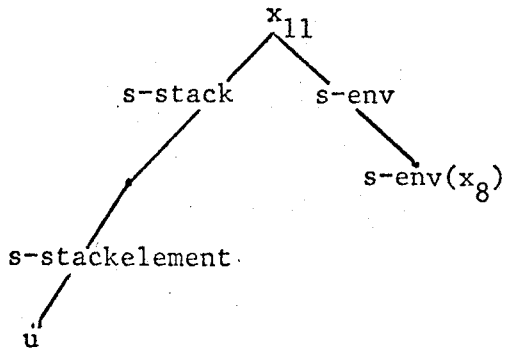
Alternative t4 is satisfied again:



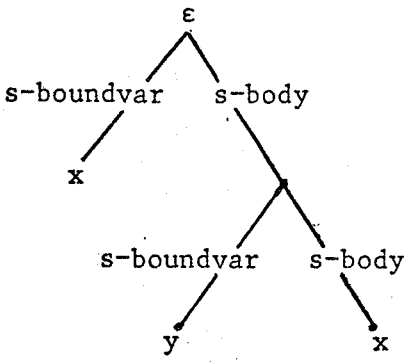
Alternative t2 implies the updating of the stack with a stack-element part yielding the value of the variable x and the insertion of a null control part, thus obtaining:



At this point alternative t1 is satisfied and we get a null control part , a null dump part and a stackelement part of the stack yielding the final value of our original expression:



We can say that the expression ε



which is the rator part of the rator part of our original expression, chooses the first of the two arguments to which it is applied.

Now, let us look at the alternative (t2); the function val may be easily given in terms of the primitives we have in the system, possibly using also the predicate "eq": OB x OB → {T,F} which applied to a pair of arguments gives as a result, the truth value true (T) if the two objects are equal and false (F) otherwise. The result of the function val (E, var) is the value presently associated in the environment E to the variable var.

We could easily avoid the use of the function val if we define the environment, i.e. the predicate is-env, in a slightly different way.

Let us consider, in fact, the following predicate definition scheme:

$$p = (\{ <s:p_o> \parallel p_1(s) \})$$

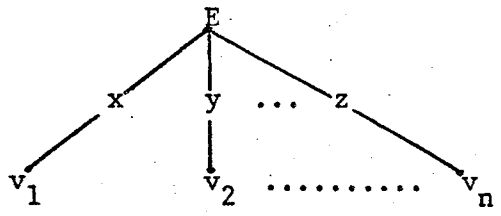
where p is the name of the predicate being defined, and p_o and p_1 are known predicates. This form allows the definition of objects sets having a variable but finite number of immediate components (instead of only two, for example, like in the is-environment predicate, AS4) of the form $<s_i:x_i>$, where s_i must belong to the set of selectors satisfying the predicate p_1 and x_i satisfies the predicate p_o.

This kind of predicate construction form is very useful in the definition of the is-env predicate, if we have in mind a representation of the environment as a table, so that val becomes a look-up into the table itself.

Rule AS4 may be substituted by:

$$\text{(AS4')} \qquad \text{is-env} = (\{ <s: \text{is-value}> \parallel \text{is-var}(s) \})$$

thus obtaining for the environment the following structure:



where $x,y,\ldots z \in S$ are selectors satisfying the is-var predicate and $v_1, v_2, \ldots v_n$ satisfy the is-value predicate.

Adopting (AS4') we simply have:

$$\text{val}\ (E,\ \text{var}) \equiv \text{var}\ (E)$$

i.e. every variable var is used as a selector for the object E, and it just selects its associated value. As we shall see later this style of table looking is also adopted in our concrete representation of the syntax and semantics of the SECD machine. Alternative t2 may be so rewritten:

t2') is-var (s-contrelement (C)) →

s-stack ← μ_0(<s-stackelement: s-contrelement (C)(E)>,
<s-stack: S>)
s-control ← SAME as t2,

and consequently alternative t4 for what concerns the s-env transformation should be so modified:

s-env ← μ(E; < s-var (s-stackelement (S)):
s-stackelement (s-stack (S))>)

2 - Concrete representation of the syntax and semantics of the SECD machine.

2.1 - Once the abstract syntax is given, for example, by means of IAS1 - IAS17 or AS1 - AS15 we should like to explain how some concrete representation may be given.
Then we have to solve the following problem: to choose some concrete data structure and to define suitable sets on this data structure according to the intuitive abstract syntax which characterize the object we are dealing with. The original formulation of the SECD machine given by Landin deals with list structures as a basis for object definition.

2.1.1 - Representation via list structures. A list can be characterized by the following definition (see [1]): a list is either null (i.e. it is the empty object) or it has a head (h), which may be an atomic object or a list, and a tail (t), which is a list . The basic operations among list structure data we shall use in our definition are the following (see [3]):
: a dyadic function, written in infix notation which constructs the list of its arguments; it corresponds to the cons function of LISP, that is a:b = cons [a; cons [b: NIL]];
h (short for head) gives the first element of a list, that is it corresponds to the car function of LISP;
t (short for tail) gives the list of all but the first element of a given list; it corresponds to the cdr function of LISP;
null is a predicate which is true iff its argument is the empty or null list, denoted by "( )";
eq is a dyadic predicate which is true iff its arguments are equal.

The definition of the objects needed for the definition

of the SECD machine may be so given in a pseudo context free notation in which the usual concatenation operator is substituted by the list constructor operator ":"; it will be seen that some classes will be, for convenience, defined as a n-tuple even though their definitions could be given by means of list structures.

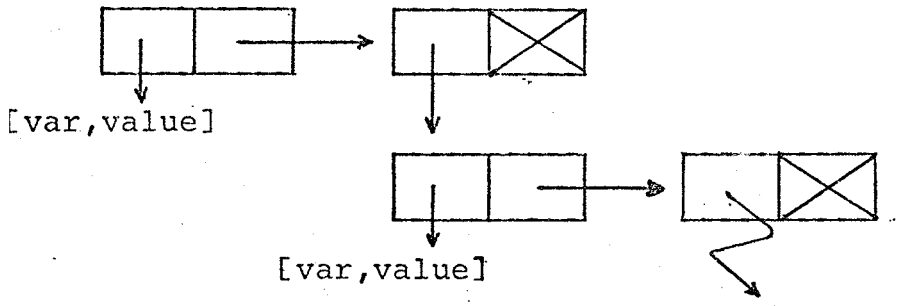
So we have:

(CL1) machine ::= [stack, env, control, dump]+
(CL2) stack ::= ( ) | stackelement : stack
(CL3) stackelement ::= closure | expression
(CL4) env :: = ( ) | envelement : env
(CL5) envelement :: = [var, value]
(CL6) control :: = ( ) | contrelement : control
(CL7) contrelement :: = expression | ap
(CL8) dump : = machine | ( )
(CL9) closure :: = env : var : expression
(CL10) expression :: = var | λexpr | combination
(CL11) λexpr :: = λ : boundvar : body
(CL12) boundvar :: = var
(CL13) combination :: = rator: rand
(CL14) rator :: = expression
(CL15) rand :: = expression
(CL16) value :: = closure | expression
(CL17) body :: = expression

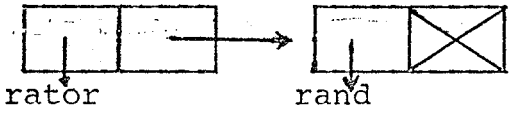
As before var is a set of atomic objects, e.g. identifiers.

+ This is a 4-tuple

For example the list env may be so visualized:



or an expression, for its combination alternative, may be so represented



Following CL1-CL17, each language may be associated with a predicate which is true if and only if its argument is an element of that language, that is if it is a list which can be generated in a like context free way according to the CL1-CL17 productions, starting with the name of the language itself. For example the predicate is-λexpr is true if and only if its argument is a list, the first element of which is the symbol λ, the second element is a boundvar i.e. a variable, the third and last element is an expression, that is it satisfies the predicate is-expression. On its turn, the predicate is-expression is true if and only if its argument is a variable, or (recursively) satisfies the predicate is-λexpr , or satisfies the predicate is-combination; and so on.

Moreover, let us consider any definition containing in its right part the operator:, for example:

(CL13) combination :: = rator : rand;

to any argument of the operator : is associated a function, i.e. a selector, which selects that part of its argument which corresponds to the argument of: By CL13, there are defined two concrete selectors, s-rand and s-rator, the

argument of which must be a list satisfying the predicate is-combination. In practice the selector s-rator is simply the function h, while s-rand is ht (that is the head of the tail of the argument). Thus, for every class name occurring in the right hand members of CL1-CL17 one can construct the corresponding concrete selector acting on appropiate list structures. On the other hand to each language considered in the syntax definition there may correspond a function which builds the elements of that language. Thus, to the language closure there corresponds the function k-closure (construct-closure) the arguments of which are an environment, a variable and an expression, and the result is the corresponding closure.

Analogously, there may exist functions k-λexp, k-combination etc. which may be defined by means of the primitive functions for list structure data.

For example, the notation λ:var:expression can be used instead of the equivalent k - λexpr (λ ,var, expression).

We remark that the important fact about the preceding predicates, selectors and constructors is that they are automatically defined as soon as the syntax definition is given.

2.1.2 - For what concerns the definition of semantics we may use explicitely these predicates, selectors and constructors and, obviously, the primitive functions on lists.

We shall use the function val [x;y] where x is an environment and y a variable and the result is the value associated in x to y; according to CL1-CL17 we may define this function in LISP terms like:

val [x;y] = [null [x] $\rightarrow$ ERROR;

eq [y; s-var [h [x]]] → s-value [h [x]]
T → val [t [x] ; y]]

Now we can give the definition of the function transform operating on a 4-tuple (machine) according to CL1-CL17, using a conditional expressions-like language:

transform (machine) =
if null (C) then [hS:S',E',C',D']
else if is-var(hC) then
[val(E,hC): S,E,tC,D]
else if is- λ expr (hC) then
[k-closure (E,s-boundvar (hC), s-body (hC)): S,E,tC,D]
else if eq(hC,ap) ∧ is-closure (hS) then
[(),(s-boundvar (hS), htS):s-env(hS), s-body(hS),
[ttS,E,tC,D]]
else if eq (hC,ap) ∧ ~ is-closure (hS) then
[hS(htS):ttS,E,tC,D]
else if is-combination (hC) then
[S,E,s-rand(hC):s-rator(hC) : ap,tC,D]

where S,E,C,D,S',E',C',D' are defined as in 1.2..

2.2 - Representation via string structures

2.2.1. Syntax definition in BNF.

It is well known that the only available primitive operation in defining string structures by means of a context free grammar is the concatenation operator, denoted by writing down contiguously its arguments. So, given a string x and a string y the concatenation operator, applied to the pair x,y is denoted by "xy".

In this way a BNF definition of the concrete syntax

following AS1-AS17 may be written as:

(CS1) <machine> :: = (<stack>∪<env> ∪ <control>∪ <dump>)

(CS2) <stack> :: = * | * <stackel><stack>

(CS3) <stackel> :: = <closure>|<expr>

(CS4) <env> :: = T|T <envel><env>

(CS5) <envel> :: = <var> ,<value>

(CS6) <control> :: = ~ |~<contrelem><control>

(CS7) <contrelem>:: = <expr> | A

(CS8) <dump> :: = :|<machine>

(CS9) <closure> :: = K (<env>∩<var>∩<expr>)

(CS10)<expr> :: = <var> | <Δexpr> | <combination>

(CS11)<Δexpr> :: = <boundvar><body>

(CS12)<boundvar>:: = <var>

(CS13)<combination>:: = (<rator> .<rand>)

(CS14)<rator> :: = <expr>

(CS15)<rand> :: = <expr>

(CS16)<value> :: =<closure>|<expr>

(CS17)<body> :: = <expr>

As before the class <var> may be defined as any <identifier>. In this case a possible expression may be a <Δexpr> (i.e. a λ expr) like:

x Δ (x.x) where the body is a <combination> of the form (<var> . <var>) and so on. Consider for example the <env> (CS4) class; the empty environment is represented by T, otherwise we have, for example,

T y, xΔ(x.x) T

when the environment contains only a <envel> where the <var> part is y and the <value> part is an <expression> i.e. a <Δ expr>. Thus we have a string of pairs separated by the symbol T; an analogous structure is given by (CS2) for the class <stack> . A <dump> is instead a string of e.g.

the following kind (remember that the null <dump> is represented by ":"):

(*5*x*UTy,x △ (x.x)TU~A~U:)

where the <stack> part is composed by two <stackel>, 5 and x respectively; the <env> part is the yet shown element, the <control> part in an A and the <dump> part is empty, i.e. represented by ":".

In the appendix it will be shown a concrete example using a syntax definition following CS1-CS17 slightly modified in order to have a total precedence syntactic recongizer (see [8]). In the actual syntax of the running example the primitive objects playing the role of a variable are distinct as it follows:

<VAR> :: = X | Y | Z | F | G | H (i.e. only 6 elements)

<OP > :: = + | - | x | / (i.e. the four elementary aritmetic operations)

<CONST>:: = "any signed or unsigned integer"

The case where an operator in a combination is an elementary arithmetic operator will be treated separately in the sense that the four elementary operations are directly available into the system by which a λ-expression is interpreted (APS algorithm, see below).

2.2.2 - Semantics definition by means of APS

Once a formal definition of the concrete syntax is given, in order to describe the transform function we need a tool for easily manipulate concrete string structures.

In a precedent paper [4] we defined an Algorithmic

Programming system (APS) primarly, in order to easily describe every string manipulation algorithm . After this theoretical foundation, APS has been implemented, using the APL language, on the IBM 360/67 computer. The basic ideas of the implementation of APS were given in [5] and an extensive account of the problem of "pattern matching" (see below), the most relevant characteristic of the system, was given in [6].

APS consists in some generalizations of the concept of a Markov Algorithm: transformation rules are used to describe algorithms and a set of transformation rules constitute an APS algorithm. The execution of an APS algorithm causes the changing of one or several object strings which match specified patterns.

A"pattern" is also called a "structure" and depends on the concrete syntax defined in the system by the user. Thus the "semantics" is specified by means of an APS algorithm whose scheme of application is the Markov one.

APS is a system which can be used for all the problems involving some "pattern matching", i.e. every problem which implies, in its solution, questions like: <u>if</u> the actual configuration of a certain string matchs a given pattern <u>then</u> transform this string in a given way <u>else</u> look for the applicability of the next rule. Tipically APS can be used as a Translator Writing System (see [9]) but it is also applicable to other problems. In APS a pattern is called a <u>structure</u> which is simply a word on the alphabet AUB', where:

A is the alphabet of terminal symbols

B is a set of metasymbols, i.e. names of languages

B' = B x N, N being the set of non-negative integers.

In APS a language is called a class and is defined in Backus Normal Form as we have just seen is section 2.2.1; not every context-free language can be treated by the present version of APS which deals only with regular and precedence languages (see [8]) ; it will be seen, in fact, that the actual syntax for the SECD machine used in our running example results slightly modified with respect to CS1-CS17.

If <name> is the name of a class, i.e. <name>∈B, then the corresponding elements of B' are noted:

<name> for (<name>, 0)
<name.1> for (<name>, 1)
<name.2> for (<name>, 2)

and so on.

If w is a string on A and $t = t_1 t_2 \ldots t_k$ is a structure (t_i belongs to AUB', for every i=1,2,...,k), we say that w contains a substring of structure t iff there exists a decomposition $w = w_o w_1 \ldots w_k w_{k+1}$, such that

S1) $t_i = t_j$ implies $w_i = w_j$
S2) $t_i \in A$ implies $w_i = t_i$
S3) $t_i = (L,n) \in B'$, implies t_i is in the class whose name is L.

A string w can contain several substrings of structure t and each substring can be subdivided in several ways in order to satisfy properties S1,S2,S3; thus we define an order in the set of all the possible decompositions; $w=w_o w_1 \ldots w_k w_{k+1}$ precedes $w=w'_o w'_1 \ldots w'_k w'_{k+1}$ iff there exists a substript j such that

for every $i < j$: $w_i = w'_i$

and $|w_j| < |w'_j|$

Now the canonical t- decomposition of w is the first decomposition of w satisfying S1,S2,S3 in the specified order.

Strings may be modified by means of transformation rules (t.r.). A t.r. consists of a left hand member and a right hand member; the left hand member specifies structures for some strings (on which the algorithm operates); if the specified strings contain substrings of the relative structure then these substrings are substituted by other strings obtained from the right hand side of the t.r., using information derived from the previous operation of "pattern matching". The application of an APS algorithm follows the Markov scheme (see [10]); however, the use of pointers (see[4]) greatly reduces the number of trials, speeding up the whole algorithm without changing the theoretical impact of the scheme. For more details see [4], [5], [6], [9] references ;these few remarks intend only to give an introduction to APS through the SECD machine example.

A t.r. acting on three object strings, say S,E,C, may be of the form:

C: ε<VAR>Δ<EXPR>$\sim \rightarrow \varepsilon$

S: $[\rightarrow [\underline{k}$(<ENV>$\cap$<VAR>$\cap$<EXPR>)

E: [<ENV>] $\rightarrow \rightarrow$

it reads as follows:

if string C (control) contains a symbol ε (pointer) followed by an element belonging to <VAR> followed by the terminal symbol Δ followed by an expression belonging to <EXPR> followed by the terminal"$\sim$" and if the string

S (stack) contains the symbol "[" and if the string E (evironment) contains the "[" symbol followed by an element belonging to <ENV> followed by the "]" symbol then delete all the matched characters after the symbol ε in the string C; in the string S, after the symbol "[" write down the sequence "* k (" followed by the element found in the string E matching the <ENV> part followed by the symbol "∩" and the element, of the string C, matching the <VAR> part followed by "∩" and the expression found in the string C matching the <EXPR> part and finally the symbol ")"; the arrow in the right hand part of E means that this string remains unchanged.

So, for example, if the actual strings were the following ones:

C: εFΔ((+.(F.3)).(F.4))~A~
S: [* k (T∩X∩((x.X).X))*]
E: [T]

i.e. a control containing a λ-expression whose bound variable is F and whose body is the combination ((+.(F.3)).(F.4)) followed by ~A~ and a stack containing the closure *k(T∩X∩((x.X).X))* and an empty environment, after the applitication of the above t.r. we get:

C: ε A~
S: [* k (T∩F∩((+.(F.3).(F.4)))* k (T∩X∩((x.X).X))*]
E: [T]

Note that this transformation rule is the exact equivalent of the point t3) of the transform function of section 1.2 and, obviously, of the third alternative of the transform function in section 2.1.2.

In the actual implementation of APS, in the right hand sides of the t.r'.s, functions can be used: these functions

may be those predefined into the system (in practice the four elementary operations, the six relations and the three basic logical operations) or they can be defined by the user; in this case the definition of these functions on LISP-like format can be introduced into the system during the "input phase".

A function can be defined in the following way:

A:<FACT>:1

means that a new function is added (A) into the system; its name is FACT and it has only a parameter.

The actual definition of the function is given by means of successive conditions; the system (recall that it is a conversational one) asks for the first condition and the user introduces it:

[1] (first condition asked by the system)

<=|<1>|0>→ 1 (typed by the user)

this condition is read:
If the first (and in this case the only) parameter, indicated by <1>, equals the string "0", then the result of the function call <FACT|0> is the string "1". We remark that we always handle strings. Then the system asks for the second condition:

[2] (second condition asked by the system)

1 → <x|<1>|<FACT|<-|<1>|1>>> (typed by the user)

that is read:

since the condition is always true (the "1" before the arrow means "T") then the result of, e.g., the function call <FACT|2> is <x|2|<FACT< -|2|1>>> .

In this case we have a recursive definition of the factorial function with the obvious meaning of x (times) and - (substraction) (which are predefined functions).

Then the system asks for a third condition and the user answers with a blank line, indicating that the function definition is complete. In the above case we would have the following computation:

<-|2|1> = 1 so
<x|2|<FACT|<-|2|1>>> = <x|2|<FACT|1>>,
<FACT|1> = <x|1|<FACT|<-|1|1>>> and since
<-|1|1> = 0 and
<FACT|0> = 1 we get
<x|2|<x|1|1>> = <x|2|1> = 2.

3 - A sketch of the APL implementation of APS.

Presently (fig. 1) the APL implementation of APS occupies two workspaces called EDITOR and EXECUTOR respectively; linkage between the two workspaces is achieved by means of a set of shared variables, which are collectively called VG (group name).

	EDITOR	EXECUTOR
SYNTAX	CLASSES EDITOR	APS
SEMANTICS	FUNCTION EDITOR	INTERPRETER
DEF.	TRANS. RULES ED.	

VG

fig. 1

The EDITOR contains all the editing programs i.e. that part of the system which allows the input of the syntax and semantics definitions for an actual problem, that is the classes, the functions (as above shown) and the APS algorithm (t.r.s); the EXECUTOR contains the universal APS algorithm, i.e. a program that interprets every APS algorithm.

The definition of the syntax of APS algorithms is accomplished by means of an edit function for syntactic classes which allows the user, in a first moment, to introduce the classes into the system, then to correct them, that is modifying them or adding other classes or deleting the existing ones.

The user has at his disposal some predefined classes; presently only five classes are predefined into the system; they are commonly used languages like <letter>, <digit>, <identifier> and so on.

When the input of the syntax is completed the system begins to elaborate the definitions in order to:

- get as much information as possible about classes and pass it to the APS interpreter (in EXECUTOR);
- build the syntactic recognizer for each class.

At this point the user can introduce the functions, as just said, and then the transformation rules constituting the body of the algorithm. The user has at his disposal an editor which allows him to add, modify and display one or several rules, and also only part of them. Any t.r. may act on one or several strings; for each string involved in the t.r. the user must introduce:

- the name of the string followed by a separator;
- the left hand member, containing the structure to be looked for in the string;
- the transformation arrow

- the right hand member, that is what has to be substituted to the substring which matched the structure in the l.h.m..

Once we have introduced the whole algorithm and possibly displayed and corrected it, we exit the t.r.s. editor.At this point, the t.r.s. are analyzed, and possibly contextual errors are signalled; if everything is all right the system renumbers all the t.r.s. in order to give to each of them an integer number as reference.

Now, we can show our APS transform algorithm which operates on the four strings S(stack),E(environment), C(control) and D(dump) whose structure is the one given by CS1-CS17.

ALGORITHM

[1]
C: ρ → ϵ<>~□
[2]
C: ϵ~ → ϵ
[3]
C: ϵ! →
[4]
C: ϵ□ → ρ
D: [:] → →
S: [<*STACK*>] → [*]
E: [<*ENV*>] → [⊤]
[5]
C: ϵ□ → ϵ<*CONTROL*>□
S: [*<*STACKEL*><*STACK*.1>] → [*<*STACKEL*><*STACK*.2>]
E: [<*ENV*.1>] → [<*ENV*.2>]
D: [:(<*CONTROL*>∪<*STACK*.2>∪<*ENV*.2>∪<*DUMP*.1>)<*DUMP*.2>] → [<*DUMP*.1>]
[6]
C: ϵ<*VAR*>~ → ϵ
S: [→ [*<*VALUE*>
E: ⊤<*VAR*>,<*VALUE*>⊤ → →
[7]
C: ϵ<*CONST*>~ → ϵ
S: [→ [*<*CONST*>
[8]
C: ϵ<*OP*>~ → ϵ
S: [→ [*<*OP*>
[9]
C: ϵ<*VAR*>Δ<*EXPR*>~ → ϵ
S: [→ [*$\underline{K}$(<*ENV*>∩<*VAR*>∩<*EXPR*>)
E: [<*ENV*>] → →
[10]
C: ϵ$\underline{A}$~$\underline{A}$~ → ϵ
S: [*<*OP*>*<*STACKEL*.1>*<*STACKEL*.2>* → [*<<*OP*>|<*STACKEL*.1>|<*STACKEL*.2>>*
[11]
C: ϵ$\underline{A}$<*CONTROL*>□ → ϵ<*EXPR*>~□
S: [*$\underline{K}$(<*ENV*.1>∩<*VAR*>∩<*EXPR*>)*<*STACKEL*><*STACK*>] → [*]
E: [<*ENV*.2>] → [⊤<*VAR*>,<*STACKEL*><*ENV*.1>]
D: [<*DUMP*>] → [:(<*CONTROL*>∪<*STACK*>∪<*ENV*.2>∪<*DUMP*>):]
[12]
C: ϵ$\underline{A}$~ → ϵ
S: [*<*STACKEL*.1>*<*STACKEL*.2>* → [*(<*STACKEL*.1>.<*STACKEL*.2>)*
[13]
C: ϵ(<*OP*>.<*EXPR*>)~ → ϵ<*EXPR*>~<*OP*>~$\underline{A}$~
[14]
C: ϵ(<*EXPR*.1>.<*EXPR*.2>)~ → ϵ<*EXPR*.2>~<*EXPR*.1>~$\underline{A}$~

Explanation of the APS transform algorithm:

[1] this rule asks for a new expression to be introduced (and then evaluated) into the control string; in the r.h.m we have in fact the symbols <> which mean "input" from the keyboard. The symbol ρ in the l.h.m is created into the string C, whenever a previous expression has been yet evaluated and it is also automatically put in by the system at the beginning of the session.

[2] Causes the deleting of the symbol $\sim$ in the control string whenever it is encountered by the pointer ϵ.

[3] The symbol! causes the complete stop of the algorithm deleting the pointer ϵ in the control string. The symbol ! has been possibly typed in by the user whenever an input is requested by means of rule [1]. It causes the end of the session.

[4] Since the evaluation of an expression has been completed (i.e. the scanning symbol ϵ has reached the end $\square$ of the control string) the pointer ρ (see rule [1]) is created in the control string, if the dump is empty; it remains unchanged and the stack and environnment strings become empty ([*] and [T] respectively).

[5] If the control is empty (as in [4]) but the dump is not empty, then put into the control string the control part of the dump, transform the stack string in such a way that the first element of S remains the first one but the other part be the stack part of the dump; and the environment becomes the environment part of the dump (this rule is the equivalent of t1 of transform function).

[6] It is the equivalent of t 2 ; the control contains a

variable , then its value, found in the environment, is set as first element of the stack.

[7] This rule treats separately the case in which an expression is a constant (in this case a numeric one i.e. the case in which we have an expression which is a variable whose value is the variable itself (e.g. 5,7, and so on); We note that this case would be avoided if we think at an environment initially containing such ccuples).

[8] The case that the expression in the control is an operator belonging to <op> (i.e. is one of the 4 elementary arithmetic operations directly available into the system) is treated separately from the general case of rule [6]; so rules [7] and [8] are particular cases of t2. As in rules [6] and [7] the <op> found in the control is put as first element of the stack.

[9] This rule refers to the case when in the control string there is a <Δexpr> (see, t3). A closure, in which are the actual environment, the boundvar and the body of the <Δexpr> are visualized, is put as first element of the stack, and the environment is unchanged.

[10] This rule is peculiar of our APS algorithm in the sense that it carries out the evaluation of the 4 dyadic elementary operations applied to the next pair of arguments in the stack. As yet noted, the corresponding operators, belonging to the class <op>, have been previously put into the stack by means of rule [8], and hence the result of the evaluation of the function call

<<op>|<stackel.1>|<stackel.2>> substitutes, into the stack, the triple <op>*<stackel.1>*<stackel.2>*. The presence, into the control string of the pair A~A~ ensures that the operator is a dyadic one (see also in the appendix the lines "trace [8]" and "trace [10]")

[11] This rule is equivalent to t4 , i.e. in the control there is an A symbol, while the top element of the stack is a closure; then the actual control, environment, dump and all the stack part, after its second element, are the new components of the dump; the expression appearing in the closure becomes the new control expression, and the environment is now constituted by the couple <var>, <stackel> (where <var> is the one coccuring in the closure and <stackel> is the element following the closure in the stack) and the environment part of the closure.

[12] Is the equivalent of t5, i.e. in the control there is A symbol , while the top element of the stack is not a closure; then we have as new top element of the stack the functional application of the first element of the stack to the second one (This is, in a certain sense , the application of every monadic function to its argument).

[13] This rule is a particular case of t6, i.e. the case where the control string contains a combination whose operator part is a dyadic arithmetic operator belonging to <op>; then we have as new first control element the operand part (in order to evaluate it before the functional application of the operator; the transform function in fact is a call by value machine), and as second element the operator part.

[14] This is the equivalent of t6; it treats the general

case of a combination whose operator and operand part are any expression.

In the appendix, a running session is reported. We are in the EXECUTOR WS and the APS program is called; in this case the "transform" algorithm is yet stored under APS. The system asks for the initial contents of the strings C,S,E, and D; the user types the contents he wants; in this case we have an empty control string (a blank line) an empty stack, environment and dump ([*], [T]) and [:] respectively). In the control string the initial pointer "ρ" is put in by the system. After these operations have been completed, the system asks whether a complete trace of the flow of the algorithm execution is desired. Typing N (for No) we have, as in this case, the trace of only the applied rules (this is visualised with the lines like TRACE[1], TRACE [14] and so on); the line [1]:C: means that the rule [1] asks for an input referring to the string C. In this case the λ- expression typed in is the one given by Wegner in [7]. At the end of the evaluation of the given expression we have a new request of input ([1]: C:) for the control string and, in this case, the unevaluable expression (also given in Wegner [7]) is typed in; as it can be seen we have an infinite loop, rules 14,6,6,11, which indicates that the expression has no normal form.

APPENDIX

```
      APS

INPUT THE INITIAL CONTENTS OF STRINGS

C:

S:
[*]
E:
[T]
D:
[:]

DO YOU WISH THE FLOW OF THE ALGORITHM?
N


COMPUTATION BEGINS


[1]:C:
(FΔ((+.(F.3)).(F.4)).XΔ((×.X).X))

TRACE [1]
C: ϵ(FΔ((+.(F.3)).(F.4)).XΔ((×.X).X))~□



TRACE [14]
C: ϵXΔ((×.X).X)~FΔ((+.(F.3)).(F.4))~1~□


TRACE [9]
C: ϵFΔ((+.(F.3)).(F.4))~A~□
S: [*K(TnXn((×.X).X))*]
E: [T]


TRACE [9]
C: ϵA~□
S: [*K(TnFn((+.(F.3)).(F.4)))*K(TnXn((×.X).X))*]
E: [T]


TRACE [11]
C: ϵ((+.(F.3)).(F.4))~□
S: [*]
E: [TF,K(TnXn((×.X).X))T]
D: [:(~U*UTU:):]
```

TRACE [14]
C: ∈(*F*.4)~(+.(*F*.3))~*A*~□

TRACE [14]
C: ∈4~*F*~*A*~(+.(*F*.3))~*A*~□

TRACE [7]
C: ∈*F*~*A*~(+.(*F*.3))~*A*~□
S: [*4*]

TRACE [6]
C: ∈*A*~(+.(*F*.3))~*A*~□
S: [*K(T∩X∩((×.X).X))*4*]
E: [T*F*,*K*(T∩*X*∩((×.*X*).*X*))T]

TRACE [11]
C: ∈((×.*X*).*X*)~□
S: [*]
E: [T*X*,4T]
D: [:(~(+.(*F*.3))~*A*~U*UT*F*,*K*(T∩*X*∩((×.*X*).*X*))TU:(~U*UTU:):):]

TRACE [14]
C: ∈*X*~(×.*X*)~*A*~□

TRACE [6]
C: ∈(×.*X*)~*A*~□
S: [*4*]
E: [T*X*,4T]

TRACE [13]
C: ∈*X*~×~*A*~*A*~□

TRACE [6]
C: ∈×~*A*~*A*~□
S: [*4*4*]
E: [T*X*,4T]

TRACE [8]
C: ϵ*A*~*A*~□
S: [*×*4*4*]

TRACE [10]
C: ϵ□
S: [*16*]

TRACE [5]
C: ϵ~(+.(*F*.3))~*A*~□
S: [*16*]
E: [⊤*F*,*K*(⊤∩*X*∩((×.*X*).*X*))⊤]
D: [:(~∪*∪⊤∪:):]

TRACE [2]
C: ϵ(+.(*F*.3))~*A*~□

TRACE [13]
C: ϵ(*F*.3)~+~*A*~*A*~□

TRACE [14]
C: ϵ3~*F*~*A*~+~*A*~*A*~□

TRACE [7]
C: ϵ*F*~*A*~+~*A*~*A*~□
S: [*3*16*]

TRACE [6]
C: ϵ*A*~+~*A*~*A*~□
S: [**K*(⊤∩*X*∩((×.*X*).*X*))*3*16*]
E: [⊤*F*,*K*(⊤∩*X*∩((×.*X*).*X*))⊤]

TRACE [11]
C: ϵ((×.*X*).*X*)~□
S: [*]
E: [⊤*X*,3⊤]
D: [:(~+~*A*~*A*~∪*16*∪⊤*F*,*K*(⊤∩*X*∩((×.*X*).*X*))⊤∪:(~∪*∪⊤∪:):):]

```
TRACE [14]
C: ϵX~(×.X)~A~□


TRACE [6]
C: ϵ(×.X)~A~□
S: [*3*]
E: [TX,3T]


TRACE [13]
C: ϵX~×~A~A~□


TRACE [6]
C: ϵ×~A~A~□
S: [*3*3*]
E: [TX,3T]


TRACE [8]
C: ϵA~A~□
S: [*××3*3*]


TRACE [10]
C: ϵ□
S: [*9*]


TRACE [5]
C: ϵ~+~A~A~□
S: [*9*16*]
E: [TF,K(T∩X∩((×.X).X))T]
D: [:(~∪*∪T∪:):]


TRACE [2]
C: ϵ+~A~A~□


TRACE [8]
C: ϵA~A~□
S: [*+*9*16*]
```

TRACE [10]
C: ϵ□
S: [*25*]

TRACE [5]
C: ϵ~□
S: [*25*]
E: [⊤]
D: [:]

TRACE [2]
C: ϵ□

TRACE [4]
C: ρ
S: [*]
E: [⊤]
D: [:]

[1]:*C*:
(*X*Δ(*X*.*X*).*X*Δ(*X*.*X*))

TRACE [1]
C: ϵ(*X*Δ(*X*.*X*).*X*Δ(*X*.*X*))~□

TRACE [14]
C: ϵ*X*Δ(*X*.*X*)~*X*Δ(*X*.*X*)~*A*~□

TRACE [9]
C: ϵ*X*Δ(*X*.*X*)~*A*~□
S: [**K*(⊤∩*X*∩(*X*.*X*))*]
E: [⊤]

TRACE [9]
C: ϵ*A*~□
S: [**K*(⊤∩*X*∩(*X*.*X*))**K*(⊤∩*X*∩(*X*.*X*))*]
E: [⊤]

TRACE [11]
C: ϵ(*X*.*X*)~□
S: [*]
E: [τ*X*,*K*(τ∩*X*∩(*X*.*X*))τ]
D: [:(~∪*∪τ∪:):]

TRACE [14]
C: ϵ*X*~*X*~*A*~□

TRACE [6]
C: ϵ*X*~*A*~□
S: [**K*(τ∩*X*∩(*X*.*X*))*]
E: [τ*X*,*K*(τ∩*X*∩(*X*.*X*))τ]

TRACE [6]
C: ϵ*A*~□
S: [**K*(τ∩*X*∩(*X*.*X*))**K*(τ∩*X*∩(*X*.*X*))*]
E: [τ*X*,*K*(τ∩*X*∩(*X*.*X*))τ]

TRACE [11]
C: ϵ(*X*.*X*)~□
S: [*]
E: [τ*X*,*K*(τ∩*X*∩(*X*.*X*))τ]
D: [:(~∪*∪τ*X*,*K*(τ∩*X*∩(*X*.*X*))τ∪:(~∪*∪τ∪:):):]

TRACE [14]
C: ϵ*X*~*X*~*A*~□

TRACE [6]
C: ϵ*X*~*A*~□
S: [**K*(τ∩*X*∩(*X*.*X*))*]
E: [τ*X*,*K*(τ∩*X*∩(*X*.*X*))τ]

TRACE [6]
C: ϵ*A*~□
S: [**K*(τ∩*X*∩(*X*.*X*))**K*(τ∩*X*∩(*X*.*X*))*]
E: [τ*X*,*K*(τ∩*X*∩(*X*.*X*))τ]

TRACE [11]
C: ϵ(*X*.*X*)~□
S: [*]
E: [⊤*X*,*K*(⊤∩*X*∩(*X*.*X*))⊤]
D: [:(~∪*∪⊤*X*,*K*(⊤∩*X*∩(*X*.*X*))⊤∪:(~∪*∪⊤*X*,*K*(⊤∩*X*∩(*X*.*X*))⊤∪:(~∪*∪⊤∪:):):):]

REFERENCES

[1] Landin P.J. - The mechanical evaluation of expressions, Computer J., January (1964)

[2] Lucas P., Walk K. - On the formal Description of PL/1, Annual Review of Automatic Programming, 6 (1970)

[3] McCarthy J., et al. - LISP 1.5 Programmer's Manual, The M.I.T. Press, Cambridge Mass. (1962)

[4] Aguzzi G., Pinzani R., Sprugnoli R. - An Algorithmic Approach to the Semantics of Programming Languages, in "Automata Languages and Programming" (M.Nivat ed.), North-Holland Pub.Co., Amsterdam (1973)

[5] Aguzzi G., Cesarini F., Pinzani R., Soda G., Sprugnoli R. - An APL Implementation of an Interpreter Writing System, in "APL Congress 73", 9-15, North-Holland Pub.Co., Amsterdam (1973)

[6] Aguzzi G., Cesarini F., Pinzani R., Soda G., Sprugnoli R. - Towards an Automatic Generation of Interpreters, Lecture Notes in Computer Science, 1, 94-103, Springer-Verlag (1973)

[7] Wegner P. - "Programming Languages, Information Structures, and Machine Organization", McGraw-Hill Book Co.,N.Y.(1968)

[8] Colmerauer A. - Total Precedence Relations, J.ACM, 17, 1, (1970)

[9] Sprugnoli R. - Compiling Expressions: An Introduction to APS, I.E.I. Research Report B73-13, Pisa (1973)

[10] Markov A.A. - "Theory of Algorithms", Moscow,USSR Academy of Sciences (1954)