UNIVERSITÉ PARIS XI

U.E.R. MATHÉMATIQUE

91405 ORSAY FRANCE

N° 100-7420

APLASM 73

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

VOLUME 3

RECHERCHES SUR APL

EDITÉ PAR P. BRAFFORT ET J. MICHEL

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, H.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

III - 0

INTRODUCTION AU TROISIEME FASCICULE

par

P. BRAFFORT

Notre intérêt pour APL est lié aux intentions même qui ont précédé à la conception du langage : développer un système formel proche de la notation mathématique et apte à exprimer simplement les algorithmes combinatoires les plus variées.

Le livre d'IVERSON contient en fait plusieurs possibilités de développement dont APL/300 puis APLSV ne sont que des réalisations particulières.

Dans la période récente les propositions de modifications - mais surtout d'extension - du langage ont été nombreuses.

Ce troisième fascicule en donne un échantillon qui nous semble significatif mais ne prêtent évidemment pas à l'exhausticité.

Le premier article est une version révisée et augmenté d'un exposé présenté à Pise à l'occasion du XVI ème meeting de la SEAS (Share Européen Association).

On s'y propose de situer le problème général d'APL et de définir un cadre théorique pour les extensions

Le second article est plus particulièrement orienté vers le problème des structures de données. Il permet d'utiliser une liaison entre les problèmes typiquement APL et les recherches de sémantique formelle comme celles développées par l'école de Vienne.

Enfin le troisième article décrit une expérience compléte de conception et de simulation d'une extension d'APL.

III.1.

APL in perspective

by

Paul BRAFFORT

1. Preliminary remarks.

This paper is a revised and expanded version of an invited paper to the XVI th meeting of SEAS (SHARE European Association) held in Pisa (Italy) 1971 [1].

My intention was, at this time, to pinpoint the peculiarities of APL viewed as a notational system, from an epistemological point of view.

Recent developments of APL as programming language have shown convincingly the adequacy of an approach of this kind and even asks urgently for a more comprehensive and systematic treatment.

We shall proceed as follows :

- we first describe the problem from a linguistic point of view with an emphasis on the triad : notation system/mathematical formalism/programming language.
- next we show how APL fits in the natural history of notation systems.
- then we consider various concepts belonging to the field of formal systems theory which happen to play some role in the development of APL.
- finally we put APL's history in perspective, with respect to the afore mentioned considerations and we offer some prognostications of the future.

Many thanks are due to K. IVERSON for his criticism of a first draft of this paper.

2. From a linguistic point of view.

In her well-known Magnum Opus on programming languages [2] Jean SAMMETT finds herself uneasy when dealing with APL.

Indeed, APL is present at two different places in the book : as "APL 360" in section 6 (on-line systems) of Chapter IV (languages for numerical scientific problems), and as "APL" in Chapter X (significant unimplemented concepts) and she says (p.715) : "... the question has been raised as to whether this is a language or a notation".

Assessing APL 360 with objectivity is certainly made difficult by the merging of the normal seductions of a very fine conversational system with the sometimes dazzling novelties of a deliberate systematics for mathematical notation.

The various aspects which concurr to make APL's appeal to a wide variety of users are more easily sorted out if one goes to the trouble of a thorough linguistic (better say " semiotic ") investigation.

Since MORRIS, semiotics has been developed along three main axes : syntax, semantics, pragmatics.

- The syntactic pecularities of APL are a consequence of its objective to be a genuine rational notations system . It is enough here to mention for

- the richness of the alphabet

explicit and systematic
- the restriction of "valence" (number of arguments) to 0,1 and 2, and

standard
the prefix notation for monadic objects.

- the absence of function precedence and the left to right association law

for parenthesis readability

- the indexing convention avoiding typographical difficultes of subscripting and syperscripting.

All these aspects of APL could be - and in part have been - put into practice in the teaching of mathematics and in the preparation of text bookas [3].

- The semantic aspects of APL are connected with the need for entities representing a large sample of mathematical objects . Here we must notice

= the variety of elementary "types" (Boolean, integer, character, etc.. some of them - being implicit.

= the structuring of objects into arrays : vectors, matrices, etc... of finite rank

= the scaling of functional precedence : variables (and constants), functions (primitive and define), and operators (such as / , , .).

- The pragmatic aspects of APL are just a manifestation of its conception as an information processing system :

= interactive facility

= "system commands" and "system functions" (I. ■ , etc...)

= "shared variables" and the very notion of a "processor" in APLSV.

This implies that "pragmatics" here is undestood as the third fundamental component of linguistics, that is "relatranship of objects of the language to users of the language and not as a lund of "ad hoc" fractial devices.

And innovation in notation was certainly not the least obstacle, despite evidences for the urgent need of a rationale, as argued in the following paragraph.

3. Birth of a Notation.

Mathematics started and came to an already high level of sophistication without any special effort on the notation problem. After all mathematical entities are concepts among other concepts and ordinary language is a natural tool for dealing with them. The very distinction between "logistics" and "arithmetics" was not clear before the time of PLATO and the use of letters a symbols for the numerals is attested not long ago B.C. But such a notational system - limited as it is - remains awkward (for example 29342 could be written M ; θτμβ' or, κθ.τμβ'(3), and one must wait for DICPHANTOS (~ 300 A.D.) to find a symbolic notation for variables as well. For him an equation which, for us, could look like

$$(x^3 + 8x) - (5x^2 + 1) = x$$

should be written, however,

$$x^{\nu}\ \alpha\ \zeta\zeta\eta\ \phi\ \delta^{\nu}\ \varepsilon\ \mu\ \delta\ \alpha \quad \zeta\ \alpha$$

which is not very transparent.

The current notational system for elementary algebra is rather recent : using x,y,z for the unknown comes after DESCARTES (1637). At the same time " + , x " are adopted. The " = " symbol comes from RECORDE (1557) but NEWTON or others used " $\propto$ " instead in 1680 and later.

If one takes the trouble of having a closer look into this evolution it becomes evident that the general trend is economy of space in writing formulas and decrease in the number of possible ambiguities.

It is interesting to notice that algebraic functions for which one tries to find an adequate symbol are restricted to monadic and dyadic ones (and this is still the case with BOURBAKI).

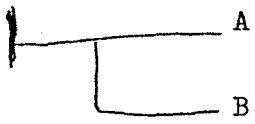
But it is still more striking to realize that, sixteen centuries after DIOPHANTOS, the standard system for algebraic notation is not completely free from ambiguities [(4)].

However, at the end of the last century and the beginning of this one, a true notational explosion took place. FREGE invited radical innovations when introducing his system of the calculus of propositions. For example

our

$$(\sim A) \not\rightarrow B$$

would be, with FREGE



FREGE's inventions are very interesting from a syntactical point of view : they show the invention of a bi-dimensional system of notation almost simultaneously with be symbolic system itself. Of course a one-dimensiona system will be preferred for reasons of typographical convenience. "Graphical" representation will nevertheless find their way is modern algebra (trees, diagrams of maps in category theory) and - decisively - in computer science !

The Polish logicians introduced later a number of interesting suggestions (and among them the famous" polish" notation (prefixing) for dya dic predicates and operations). In particular LESNIEWSKI developed an interesting ideographic system for his logics, including a systematics for the 16 binary predicates ; one has, for example,

ϕ stands for coimplication : true if and only if its arguments have the same value, both being true or both false, and so coimply each other.

$\circ$ Disjunction : true if and only if its arguments are disjunctive, exactly (i.e., at least and at most) one being true, the other false.

♀ Conjunction : true if and only if its arguments are conjointly true.

♁ Exclusion : true if and only if at most one of its arguments is true, excluding the other, which is false ..." etc...

FREGE's and LESNEWKI's systems remained unused, but nobody objected to the proposals as such. Reductance to innovation came oftem from the field of applications. One remembers the hostility of many physicists to vectorial notation. LORENTZ had to argue at length before using it for decribing MAXWELL's equations !

PEANO and his school during the period of 1889 - 1906 made a decisive effort to set up a complete and rational system of notation, introducing in particular many of the symbols of modern logics and set theory. As BURALI-FORTI says : -

" The logical symbolism presents itself under two distinct aspects ; as an abbreviated writing or tachygraphy, and as a powerful instrument for analyzing ideas, their logical relation and their development." [(5)]

The main purpose of these authors is to set up a system which gives us an economy in writing and a security of understanding. One can cite here PADOA (6)

"Cependant - tandis que l'idéographie algébrique, étant composée de signes, est arrivée, relativement, en peu de temps à un si haut degré de perfection et d'universalité - l'idéographie géométrique, étant composée de mots et entravée par les exigences philologiques et par une tradution millénaire, est

restée nationale et souvent ambigué dans une même langue. De sorte que, lorsqu'on veut construire une idéographie nouvelle, il est préférable d'avoir recours à des signes, brefs et universels, au lieu de gaspiller son temps à analyser, débattre et sanctionner la signification des mots ; c'est pourquoi l'idéographie logique a été composée de signes plutôt que de mots".

4. Names, types, structures, orders, etc...

The notational idea in APL is simply to stick to current mathematical practice as far a coherent one is already at work, and to suggest novelties only for the sake of rationality.

But granted that a solution has been found for the problem of form we are faced with ahuge problem of content .

The modern- axiomatic - usage in mathematics uses structures to define and study formal objects. The emphasis is on cartesian product (or power) and functional mapping ("application"). This implies use of a basic set and escalation over it.

In a célebrated manuscript : ψαμμ'τηs (the sand reckoner) ARCHIMEDES, in order to show that very large, but finite aggregates of finite objects are countable, developed a technique of enumeration which is not new as far as notation is concerned - because ordinary words are still used - but

shows a system at work[(7)].

The arithmetic of this time having names for numbers up to $A = 10^8$ (a myriad of myriads) and not more, ARCHIMEDES proposed to define two new concepts : "orders" and "periods".

The first period has got A orders : 1 st order is made of integers from 1 to A

2 nd order is made of integers from A to A^2

A th order is made of integers from

$$A^{A-1} \quad \text{to} \quad A^A = B$$

The second period will go the same way from A to B^2 and so on till the A th period which will give the possibility to reach B^B which is a very large sum indeed.

This is a very neat example of escalation over a basic set : here the finite set of integers from 1 to A .

The challenge of rendering ARCHIMEDES' idea in a formal system is not met by APL, but the concept of an array is a partial answer

while the use of arrays satisfies the need for cartesian product, mapping are realized through primitive and defined functions. As a matter of fact function definition looks very much like CHURCH's (after RUSSELL) notion of functional abstraction . Only the syntax differs.

All this boils down to the following concepts :

a) Entities.

There are only two entities in APL 360 : data and functions (this is well in evidence in[(8)]). It is enlighting to examine to which point they are similar or dissimilar :

- data and functions can be primitive or defined

primitive and defined data are respectively the so-called "constants" and "variables".

Primitive entities are presented (for input and output) as special characters from the APL character set but primitive functions are always expressed by one symbol only (while an integer will use up to 16 decimal symbols).

Defined entities will be named via an identifier which is a word in the alphanumeric subset of the APL character set.

But the specification which gives such an identifier its meaning comes, for defined data, from the assignent operator : " ← " ,and, for defined functions from a complex arrangement including the entering into definition mode via the " " operator, the special "header syntax" etc...

- data and functions are diversely connected to the four basic sets :

N = { integers $< 10^{16}$ in absolute value}

Q = { rational number $< 7.10^{75}$ in absolute value}

B = {0,1}

A = {APL accepted character set}

(the numérical values are, of course, implementation dependant).

On one hand the concept of "valence" establishes a correspondence between data and functions (a datum can be viewed as an "anadic" function).

On the other hand data sets may be "escalated", that is, one may take date from N^p, Q^r et. which means vectors if you consider the components as such, or <u>array</u> of rank n if you write

$$p = p_1 \times p_2 \times p_3 \times \ldots \times p_n .$$

This way of building complex objects from simple ones by taking cartesian products is usual in mathematics. But then one loses again parallelism between data and function except for the special case of the primitive functions

$$\phi, \ / \ ,$$

which can be viewed as vectors. It is worthwhile to notice that in his pre-implementation book [(9)], IVERSON used a matrix-like primitive function

$$\circ \qquad \begin{matrix} \vee & \wedge \\ \neq & = \end{matrix}$$

b) Orders

Another point of view, when considering the relationship between data and functions comes from the concept of order.

If you consider data as belonging to the lowest level :order 0, and functions to order 1, there is a rational tendency to look after entities of higher order.

Such entities exist indeed in APL 360 but with some peculiarities of their own :

- the concept of "inner product" can be interpreted as the implementation of a "order 2" dyadic primitive function, represented by the symbol "." , the argument of which are the so-called "dyadic scalar primitive functions" (which are or course of order 1) ;

- the concept of "outer product" can be interpretéd as the implementation of a "order 2" monadic primitive function, represented by the sequence of symbols "∘." , the argument of which is a dyadic scalar primitive function ;

- the concept of "reduction" can be interpreted as the implementation of a "order 2" monadic primitive function, represented by the symbol "/" , the argument of which is a dyadic scalar primitive function.

The only trouble is that "∘." is made of <u>two</u> symbols, and, what is more regrettable, / has got to put its argument on its left in contradiction to the regular syntax of APL monadic functions (+/v, when v is a vector is equivalent to

$$\sum_{I=\text{origin}}^{I=\text{length of } v} v(I)$$

c) Extension to arrays

Extension to arrays of scalar functions is straightforward.

But this is just a case where traditional notation satisfies itself with insufficient rigour and it could be interesting to go into more details about it.

If σ is the symbol used for a primitive "scalar" dyadic function (such as $+$, $\div$, Γ , etc...), the current practice in mathematics is to use the same symbol for the function $(E \times E \rightarrow E)$ than for the function $(E^n \times E^n \rightarrow E^n)$ defined by the well-known canonic correspondence.

But if σ is in fact a name for a special subset of $(E \times E) \times E$, this is certainly an abuse of language to use the same name for a subset of $(E^n \times E^n) \times E^n$, even if there is a standard link between the two.

Therefore it should be worthwhile to make the distinction explicit between the symbols for a primitive function when the arguments are scalars or arrays of various ranks.

This could be done by letting primitive functions be themselves considered as arrays. Then

$X \ \sigma \ Y$, when X and Y are scalar, would become

$X \ \sigma \ [\rho X]Y$ when or Y are conformable arrays of a non-null rank.

This introduces again the idea of ranking and dimensioning primitive functions, bringing them closer to primitive data.

All these observations indicate the presence, in the conception of APL objects, of a number of attributes which remain incompletely explicitated and are not all reachable form the user :

= the structural attribute are dimension and rank - but this leave aside lists and trees.

= the "type" attribute remains implicit (boolean, character, integer, decimal) but can be reached indirectly (using I.22).

= for non constants objects a name attribute is provided (which covers variables, defined functions, , etc...), the sorting of which implies other attributes.

= a new attribute appear with APLSV : this is sharing which indicates when an object is reachable by more than one user (at the same moment).

So we understand that a variety of formal (or formalizable) concepts is attached to APL objects . Some of those concepts are familiar to the user of mathematical notation, some are not. But in any case there is a strong incitation to carry over here the trend toward systematization and rationalization. This must certainly be the main guide for future extensions and implementation of the language.

5. From APL to Naples

* APL is not the last word for ever in notation or language research, and modifications are already being offered by authors and considered by implementors ([10],[11],[13]) .

While it is essential to maintain a reasonable stability, for the security of users (this implies an emphasis on extensions against modifications experiments are needed which should open new ways (whence the acronym : New APL Experimental System).

But such experiments should be conducted in accordance with the fundamental objective, which were present at the very beginning of the conception.

- on one hand, to remain close to ordinary mathematical notation (with possibly some improvement in the coherence of the notation itself) means introducing new types, new structures and further a capacity for defining types and structures.

The concepts of type, structures etc... could be embedded in a more general notion of type similar to the notion used in mathematical logic (for example in the typed lanbda-calculus). This would imply a systematisation of the notion of functional . We have pointed out that APLSV "operators" are functionals, but primitive ones. We could very well need in the future

user-defined functionals

- on the other hand we could question the usefulness or even the correctness of an approach which makes users ignorant of the system which supports the language.

This brings us back to the linguistic aspects of APL . It is now customary to refer to the traditional semiotic trinity : syntax, semantics, pragmatics [16].

- The striking syntactical feature of APL is simplicity : " valence" restricted to 2 , left association and mode dichotomy (execution, definition).

- The semantics is certainly unique by its richness as compared to programming languages, and even of standard mathematical formalisms

- Definition methods currently used for programming language semantics could be significantly improved with APL.

It is a normal practice, indeed, to take advantage of an already known language or formal system in order to "program" the entities to be analysed. It is even more fashionable to "bootstrap" the whole process by writing an interpreter for the language in terms of a smal subset of itself - subset to be accepted as sufficiently evident. This has been done for APL by LATHWELL and MEZEI in [(14)] Another line makes use of a small (metalanguage" also supposed to be sufficiently transparent. This is done

for APL by ABRAMS [(15)].

In each case unanalysed elements remain in the semantics - especially the interpretation process itself, and the pragmatics of the language is not even touched.

Therefore it is worth noting the importance of the "execute" function, together with other peculiarities which are on the borderline between semantics and pragmatics. In particular, the "carriage return" signal, corresponding to a special key on the terminal key-board, is to be viewed as a character among the other "normal" characters. Used in conjunction with execute, one finds here a facility for a complete rationalisation of the whole semiotics of the system.

A simple example will help here :

It is well known that the family of ACKERMANN functions (the first members of which are addition, multiplication, exponentiation, tetration, etc) may be generated from addition by primitive recursion. Thus, if (n) representents the n^{th} nember of the family, one has :

$$x\,(n)\,y = (x\,(n)\,y - 1)\;(n-1)\;y$$

Another possibility is to use an algorithmic definition including locps.

Now let us see how we can deals with this problem :

a. in APL 360 (XM6), ① ↔ + , ② ↔ × , ③ ↔ *

and ④ can be build without loop or recursion by × ④ y ↔ */y f ×

b. The whole ACKERMANN family is obtained inductively : if ⓝ has been defined as ACKN , one has

```
∇      Z  X(ACKN + 1) Y
[1]    Z  (ACKN)/Y X  ∇                                    (1)
```

(supposing that the reduction operator is extended to defined functions).

c. If the "execute" operator is available, it is possible to show that 5 ⍎ ('3x (y - 1) ⍴'(x/' , 'x', ('3x(y-1)⍴'⍴x)' and proceed inductively from there as in the preceding case. J. BROWN has asked whether it would be sufficient to define ⑥ in a closed, non recursive form. [12]

The answer is yes : using ⍎ makes it possible to define not only ⑥ , and inductively ⓝ from (n - 1) , but even to define directly x ⓝ y without any loop or recursion; This is a unique example of a non-recursive definition of a truly general recursive function.

A function such as "execute" is certainly of a different kind than "plus", "drop", or even "assign". It is not possible to describe its effect by mean of a mathematical object : functional application or explicit definition "Execute" obtains his meaning by reference to the APL interpreter itself which is - after all - an APL object from a "system" point of view , but a hidden one.

With APLSV many system functions are also introduced. The very concepts of shared variables and of auxiliary processor ring the (precedently ignored) entities of the system accessible to the language user. No doubt we must proceed in this direction but here the main problem is to keep this development in harmony with the first constraint mentioned : compatility with the spirit of mathematical notation.

An experiment of this kind is in progress and will be described elsewhere [17].

REFERENCES.

[1] P. BRAFFORT Soc. of SEAS XVI 1971 p.55

[2] J. SAMMETT Programming languages Prentice Hall 1969.

[3] K. IVERSON Elementary functions SRA 1969.

[4] K. IVERSON Colloque APL IRIA 1971 p.12

[5] C. BURALI-FORTI, Logica Matematica, Hoepli, 1919 , p.XIX.

[6] A.PADOA, Le logique déductive, Gauthiers-Villars, 1912, p.11

[7] S. DELSEDIME, Archives for history of exact sciences, 6 , 1970, p.345.

[8] S. PAKIN, APL 360 Reference manual, SRA, 1968.

[9] K. IVERSON, A programming language, I. WIley, 1962, p.246.

[10] A. McEWAN and P. WATSON, Quot. Quad. 2,2 , 1970, p.11

[11] J. RYAN, Quot. Quad 3 , 1971 , p.8

[12] J.A. BROWN, Quot. Quad. 2, 1, 1970, p.4

[13] APL Congress 73 North Holland 1973. cf. especially the papers by EDWARDS, VASSEUR, etc...

[14] R.H. LATHWELL, J.E. MEZEI, colloque APL, IRIA, 7-10 Sept.1971, p.181.

[15] Ph. ABRAMS, An APL machine, Report no. SLAC-114, Stanford 1970.

[16] H. ZEMANEK Com. ACM 9 1966 p.139.

[17] P. BRAFFORT and J. MICHEL . APL X : an experiment in language extensibildy Note ECSTASM N° 1975.

III.2.

EXTENSION TO APL DATATYPES WITH AXIOMATICALLY DEFINED VIENNA OBJECTS

BY

A. OLLONGREN

UNIVERSITÉ DE LEIDEN

GROUPE DE PROGRAMMATION THÉORIQUE

PREFACE

The following is intended as a contribution to the symposium APLASM (APL applied to Symbol Manipulation) to be held on December 20 and 21 1973, in Université de Paris-Sud, Centre d'Orsay, Mathématique. Owing to other commitments the author is unable to present the paper in person.

The paper consists of two sections. In the section headed "Abstract data structures" the abstract set of objects used in the Vienna definition method is introduced; its properties are defined by means of a system of axioms and a linear notation is established for the members of the set; it is well-known that the objects themselves can be represented by labeled, rooted directed trees. In the second section with the title " APL representation of objects" the representation problem of the members of the general class in APL is considered. The following suggestions for extensions to APL are introduced:

- nomination of selectors and elementary objects
- specification of composite selectors
- specification of μ mapping
- specification of selection.

Using these suggestions the Viennese linear notation for objects is easily transcribed into APL . Examples are given. As a result means and techniques for the discussion of semantics of computing processes become available in APL .

ABSTRACT DATA STRUCTURES

Computational processes are concerned with the manipulation of data, be it scalars (numbers, characters etc.), sequences of scalars (strings), arrays and so on. It is useful to introduce a general class of data structures, which contains all of the data needed for dataprocessing. We call the class a class of abstract data structures, or objects because their properties are given by a system of axioms and the representation problem is only considered afterwards.

Axioms for objects

Let $(\mathcal{O}, S, \circ)$ be a system of objects, selectors and an operation for which the following is supposed:

- S is a finite non-empty set
- $\mathcal{O}$ contains a finite non-empty set $\mathcal{E}$

Let $(S^*, \circ, I)$ be the free monoid generated by S in the usual way with $\circ$ as the group operator and I as the identity element. S^* is called the set of composite selectors. Finally let $\circ$ also be a relation between $S^* \times \mathcal{O}$ and $\mathcal{O}$ with some special proporties to be discussed presently. For the system $(\mathcal{O}, S, \circ)$ eight axioms are chosen:

A1. $s \circ A \in \mathcal{O}$ ----------(closure under selection)

A2. $(\kappa \circ s)\, A = \kappa\,(s(A))$----------(composite selection)

A3. $I \circ A = A$ ----------(identity operation)

A4. $(\exists\, \omega)(\forall\, s)\; s \circ \omega = \omega$----------(existence of null object)

A5. $(\forall\, \omega)\left[(\forall\, s)\; s \circ \omega = \omega \Rightarrow (\forall\, A)(\exists\, \kappa)\kappa \circ A = \omega\right]$
----------(composite selection of null objects)

A6. $(\forall\, \kappa, e)\left[\kappa(A) = e \Leftrightarrow \kappa(B) = e\right] \Rightarrow A = B$
----------(equality)

A7. $(\forall\, A, \kappa, e)(\exists\, B)\left[\kappa(B) = e \wedge (\forall\, \tau)\left[\neg\, \mathrm{dep}(\kappa,\tau) \Rightarrow \tau(B) = \tau(A)\right]\right]$
----------(existence of constructed object)

A8. $\mathcal{O}$ is the smallest set including the null objects and the elementary objects such that axioms 1-7 hold.

In above formulas $A, B \in \mathcal{O}$, $e \in \mathcal{E}$, $s \in S$, $\kappa, \tau \in S^*$ and ω is a null object. The dependency relation dep is defined as follows

$$\mathrm{dep}(\kappa,\tau) = (\exists\, \sigma)\left[\kappa = \sigma \circ \tau \vee \tau = \sigma \circ \kappa\right]$$

with $\sigma \in S^*$.

Discussion

From axioms 1 and 2 it is seen that $\circ$ can be regarded upon as a relation between $S^* \times \mathcal{O}$ and $\mathcal{O}$ (i.e. a non-empty subset of this Cartesian product) with the following property:
for every $(\kappa, A) \in S^* \times \mathcal{O}$ there exists a unique object B such that $(\kappa, A, B) \in \circ$. The relation $\circ$ can be regarded upon as a selection operation: for every $\kappa \in S^*$ and $A \in \mathcal{O}$ a unique B is selected. In other words: the object A has structure in general and given a composite selector κ the component B of A is selected. We write for this operation $\kappa \circ A$, $\kappa(A)$ or κA . Note that $\circ$ plays a double role as it is also the group operator in the monoid $(S^*, \circ, I)$.

Theorem 1.

There is exactly one null object.

Proof: Suppose that $\omega_1 \neq \omega_2$ satisfy axioms 1-5. Axiom 5 states that there is a composite selector κ such that $\kappa \circ \omega_1 = \omega_2$; if $\kappa = I$ we have an immediate contradiction, if $\kappa \neq I$ we get a contradiction using axiom 4.

Definitions ($s \in S$ as usual)

- The unique null object is denoted by Ω
- $\mathcal{A} = \{a \mid (\exists\, s) s \circ A = \Omega\}$ is called the set of atoms
- $\mathcal{E} = \{e \mid e \neq \Omega \wedge (\forall\, s) s \circ e = \Omega\}$ is called the set of elementary objects
- $\mathcal{C} = \mathcal{O} - \mathcal{E}$ is called the set of composite objects.

For a composite object $A \neq \Omega$ there is at least one selector s such that $s(A) \neq \Omega$.

Theorem 2.

If $A, B \in \mathcal{O}$ and $\kappa \in S^*$ such that $\kappa(A) = \kappa(B) \neq \Omega$ then
$(\forall\, \tau)[\neg \mathrm{dep}(\tau, \kappa) \Rightarrow \tau(A) = \tau(B)] \Rightarrow A = B$

Proof: If $\tau \in S^*$ such that $\tau(A) = e$ then $\tau(B) = e$ because
(a) if $\neg\, \mathrm{dep}(\tau, \kappa)$ then from $\tau(A) = \tau(B)$ follows $\tau(B) = e$
(b) if $\mathrm{dep}(\tau, \kappa)$ then
(b_1) $\tau = \xi \circ \kappa$ and from $\kappa(A) = \kappa(B)$ follows that $\tau(B) = e$
or (b_2) $\kappa = \xi \circ \tau$ and if $\xi = I$ then $\tau(B) = \kappa(B) = \kappa(A) = \tau(A) = e$;
if $\xi \neq I$ then $\kappa(A) = \Omega$ which contradicts the assumption.
As a result of this:
if there is no τ such that $\tau(A) = e$ then there is no τ such that $\tau(B) = e$

From axiom 6 we can now conclude the validity of the theorem.

Theorem 3.
The object B which satisfies axiom 7 is unique.

Proof: Suppose that B_1 and B_2 satisfy axiom 7. Then $\kappa(B_1) = \kappa(B_2) = e$ and from $(\forall \tau)[\neg\, dep(\tau,\kappa) \Rightarrow \tau(B_1) = \tau(B_2)]$ and theorem 2 we conclude that $B_1 = B_2$.

Definition
$\mu : \mathcal{O} \times S^* \times \mathcal{E} \to \mathcal{O}$ is a total mapping where the value of $\mu(A,\kappa,e)$ is the (unique) B satisfying axiom 7.

Theorem 4.
For any $B \in \mathcal{O} - \{\Omega\}$ there exists a finite sequence $B_i = \mu(B_{i-1},\kappa_i,e_i)$ $i = 1,2,\ldots,n \geq 1$ such that $B_n = B$.

Proof: Choose $B_o = \Omega$. If $B = e$ then $B = \mu(\Omega,I,e)$ and the theorem is proved. If $B \notin \mathcal{E}$ then there exists at least one selector s and at most a finite number of selectors such that $s(B) \neq \Omega$; further there exists at least one composite selector κ and at most a finite number of composite selectors such that $\kappa(B) = e$; both statements are a result of axiom 8. Suppose that $\kappa_i(B) = e_i$ $i = 1,2,\ldots,n \geq 1$. For $i \neq j$ we have $\neg\, dep(\kappa_i,\kappa_j)$. If B_n is defined by $B_i = \mu(B_{i-1},\kappa_i,e_i)$ $i = 1,2,\ldots,n \geq 1$ with $B_o = \Omega$, then we can prove with axiom 6 that $B_n = B$. This proves the theorem.

Definition
For any $B \in \mathcal{O}$, the characteristic set associated with B is
$\overline{B} = \{\langle \kappa_i : e_i \rangle \mid 1 \leq i \leq n\}$
with κ_i, e_i as defined in theorem 4. If $B = \Omega$ then $\overline{B} = \{\}$.

Theorem 5.
$A = B \Leftrightarrow \overline{A} = \overline{B}$

Proof: Trivial with above definition and axiom 6.

Theorem 6.
If $\langle \kappa_1 : e_1 \rangle, \langle \kappa_2 : e_2 \rangle \in \overline{B}$ then $\neg\, dep(\kappa_1,\kappa_2)$.

Proof: Trivial. The condition $\neg\, dep(\kappa_1,\kappa_2)$ is called the characteristic condition.

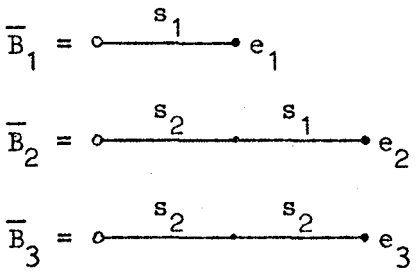
Theorem 7.

If $Z = \{\langle \kappa_i : e_i \rangle | 1 \leq i \leq n\}$ is a set for which the characteristic condition is fulfilled, then there exists $B \in \mathcal{O}$ such that $\overline{B} = Z$.

Proof: By induction, too lengthy to reproduce here.

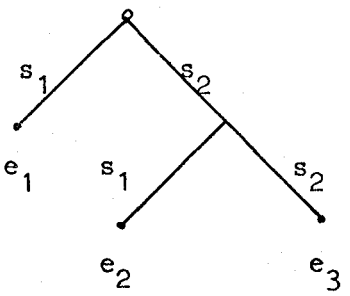
We make now a few remarks on the representation of members of the class of axiomatically defined data structures. With each object B is associated $\overline{B}$, i.e. a set of pairs. If for example $B = B_3$ where
$B_i = \mu(B_{i-1}, \kappa_i, e_i)$ $i = 1,2,3$ with $B_o = \Omega$, $\kappa_1 = s_1$, $\kappa_2 = s_1 \circ s_2$, $\kappa_3 = s_2 \circ s_2$
then
$\overline{B} = \{\langle s_1 : e_1 \rangle, \langle s_1 \circ s_2 : e_2 \rangle, \langle s_2 \circ s_2 : e_3 \rangle\}$ and $\overline{B}$ can be represented in its turn by either three rooted, directed, labeled trees without bifurcations, or as one rooted, directed, labeled tree. We have in figures

$\overline{B}$ =



or

$\overline{B}$ =



in which $\circ$ indicates a root and $\bullet$ a leaf. The labeling is done as follows: edges carry selectors as labels (each two outgoing edges of a node carry mutually distinct labels), leaves are labeled with elementary objects, no other nodes are labeled.

Theorem 8.

If the characteristic condition holds for two sets
$X = \{\langle \kappa_i : e_i \rangle | 1 \leq i \leq n\}$ and $Y = \{\langle \kappa_i' : e_i' \rangle | 1 \leq i \leq m\}$
and if $\kappa \in S^*$ then the characteristic condition holds for
$Z = \{\langle \tau : e \rangle | \langle \tau : e \rangle \in X \wedge \neg \operatorname{dep}(\kappa, \tau)\} \cup \{\langle \tau \circ \kappa : e \rangle | \langle \tau : e \rangle \in Y\}$

Proof: straight forward using theorem 6.

Definition (extension μ function).
For any $A, B \in \mathcal{O}$ and $\kappa \in S^*$ the unique object C for which
$\overline{C} = \{\langle \tau : e \rangle | \tau(A) = e \wedge \neg \operatorname{dep}(\tau, \kappa)\} \cup \{\langle \tau \circ \kappa : e \rangle | \tau(B) = e\}$
is denoted by $\mu(A; \langle \kappa : B \rangle)$

Theorem 8 states that there exists an object C with the characteristic set as shown and that it is unique. If $B = e$ then $\mu(A; <\kappa : e>)$ satisfies axiom 7 so that it is justified to use the function name μ in above definition.
Note that $\mu(A; <I : B>) = B$. Further
$\mu(A; <\kappa : \Omega>) = A$ if and only if $(\forall \tau)\ dep(\kappa,\tau) \Rightarrow \tau(A) = \Omega$.

In order to illustrate the use of the extended μ function we can write for above example

$$B_1 = \mu(\Omega; <s_1 : e_1>) \qquad \overline{B}_1 = \{<s_1 : e_1>\}$$
$$B_{12} = \mu(\Omega; <s_1 : e_2>) \qquad \overline{B}_{12} = \{<s_1 : e_2>\}$$
$$B' = \mu(B_1; <s_2 : B_{12}>) \qquad \overline{B}' = \{<s_1 : e_1>, <s_1 \circ s_2 : e_2>\}$$
$$B = \mu(B'; <s_2 \circ s_2 : e_3>) \qquad \overline{B} \text{ as shown before}$$

Definition (further extension μ function)

$$\mu(A; <\kappa_1 : B_1>, <\kappa_2 : B_2>, \ldots, <\kappa_n : B_n>)$$
$$= \mu(A; \{<\kappa_i : B_i> \mid 1 \leq i \leq n\})$$
$$= \mu[\mu(A; <\kappa_1 : B_1>); <\kappa_2 : B_2>, \ldots, <\kappa_n : B_n>]$$

For $n = 0$ $\mu(A;) = \mu(A;\{\}) = A$; for $n \geq 1$ we have a recursive definition.
For $A = \Omega$ we write $\mu_o(\ldots)$ instead of $\mu(\Omega;\ldots)$.

In above example we have
$B = \mu(\Omega; <s_1 : e_1>, <s_1 \circ s_2 : e_2> <s_2 \circ s_2 : e_3>)$
but also
$B = \mu_o\ (<s_1 : e_1>, <s_2 : B_{12}>, <s_2 \circ s_2 : e_3>)$
with B_{12} as given above. Note that $\mu(B; <\kappa : B>)$ is a ligitimate expression; the value of it is B if and only if $\kappa = I$ or $B = \Omega$; if $\kappa = s$ for instance the value of it is a new object C with the property that $s(C) = B$.

APL REPRESENTATION OF OBJECTS

It is obvious from the axioms given above that one must have two sets in advance if one considers applications: the set of selectors and the set of elementary objects. It is suggested that individual selectors and elementary objects are nominated in APL as follows (we give a few examples)

```
α'S'
α'E1'
α'ELEM','1'
```

Here α is some suitably chosen APL symbol. For the moment we need not distinguish between selectors and elementary objects.

Using above "type declaration" we can specify composite selectors. If $S1$ and $S2$ have been nominated and so exist in the system we can specify for instance

```
K ← S1 ∘ S2
P ← S1 ∘ S2 ∘ S2
Q ← S1 ∘ S2 ∘ α'S3'
```

and we introduce here the identity selector I by the specification
$I \leftarrow \alpha''$

Next we consider the problem of building objects from the nominated selectors and elementary objects. If the objects A and B have been built and if K is a composite selector then we write for the μ mapping $\mu(A;\langle K B\rangle)$ in APL

```
A K B
```

So that K is considered to be a dyadic operator. Each object A is characterised by its characteristic set $\overline{A}$; it can be nominated as an elementary object by

```
α A .
```

The null object could be given a special APL character, but it is suggested to introduce first a selection specification and afterwards specify the null object instead.

If A is either an elementary object or a composite object built by a sequence of μ mappings, and if K is a composite selector, then $K(A)$ is again an object by axiom 2. It is proposed to use the compression operation in APL and to write

$$B \leftarrow K/A$$

With axiom 3 we have then for any object A .

$$B \leftarrow I/A \leftrightarrow A \leftarrow A$$

However if E is an elementary object and K is not I , $K(A)$ is the empty object, and so we can give it a name (for instance 0) in APL by

$$0 \leftarrow K/E$$

Note that $\alpha\, 0$ is then the characteristic set of the empty object (i.e. the empty set) and that this is not equal to α'' . Since we do not need a special character for the null object it is reasonable to write for
$A = \mu(\Omega;<K:B>)$ in APL

$$A \leftarrow K\ B$$

Finally we need an APL notation for the extended μ mapping in which the μ function is given more than three arguments. It is proposed that the result of the following statements

$$K \leftarrow K1,K2,K3$$
$$B \leftarrow B1,B2,B3$$
$$C \leftarrow A\ K\ B$$

(where either the K's are nominated selectors or composite selectors and the B's are objects) is $\mu(A;<K1:B1>,<K2:B2>,<K3:B3>)$.

APPLICATION

We consider only binary arithmetic expressions in this section. An expression E in this class has three components: $SOP1(E)$ and $SOP2(E)$ are variables, constants (elementary objects) or themselves binary expressions and $SOP(E)$ is a binary arithmetic operator (an elementary object).

So for the expression E = a + b ⋆ c we have

$SOP1(E)$ = a $SOP(E)$ = + $SOP1 \circ SOP2(E)$ = b
$SOP2(E)$ = b ⋆ c $SOP \circ SOP2(E)$ = ⋆ $SOP2 \circ SOP2(E)$ = c

and written as a μ function

$$E = \mu_o(\langle s\circ p_1 : a\rangle, \langle s\circ p_1 \circ s\circ p_2 : b\rangle, \langle s\circ p_2 \circ s\circ p_2 : c\rangle, \langle s\circ p : +\rangle, \langle s\circ p \circ s\circ p_2 : \ast\rangle)$$

For above expression we can write using the extensions to APL suggested above

```
α'SOP'
α'SOP1'
α'SOP2'
K1 ← SOP1 ∘ SOP2
K2 ← SOP2 ∘ SOP2
K ← SOP ∘ SOP2
E ← SOP, SOP1, K1, K2, K  (α'+'),(α'A'),(α'B'),(α'C'),(α'⋆')
```

We have then the following structural properties of E

```
SOP/E ↔ +
SOP1/E ↔ A
```

```
E2 ← SOP2/E ↔ E2 ← SOP, SOP1, SOP2 (α'⋆'),(α'B'),(α'C')
```

If $E1$ and $E2$ are two binary expressions then

```
E ← SOP, SOP1, SOP2  (α'+'),E1,E2
```

is a new binary expression. If $E2$ is as above and $E1 \leftarrow$ α'A' then E as above is retrieved.

ACKNOWLEDGEMENT

The idea of extending APL to accomodate the tree structures of the Vienna definition method is due to Mr. P. Sipos. Discussions with Mr. Sipos have helped to cast above suggestions into what is hoped to be a reasonable form.

LITERATURE

A. Ollongren 1974, "Definition of programming languages by interpreting automata", Acad. Press APIC Series no. 11, to appear.

P. Lucas & K. Walk 1969, "On the formal description of PL/I", Annual Review in Automatic Programming, Vol. 6, Part 3, p. 105 - 182.

APL - GA: an immediate extension of APLSV.

J. MICHEL

C.N.R.S. (France).

Résumé : l'évolution des systèmes APL conduit naturellement à des extensions dont l'une, celle des structures de données en "tableaux de tableaux" a déjà fait l'objet de plusieurs propositions. APL - GA est une extension de ce type qui aborde également d'uatres problèmes significatifs : types, compilation, etc... Un programme APLSV permettant de simuler APL-GA est produite, et plusieurs exemples d'applications sont analysés.

1. INTRODUCTION. APL has been conceived towards 1960. During almost fifteen years many development occured which opened new directions to the language and still maintained the original spirit.

Since the appearance of Iverson's book, several extensions and modifications have been proposed which are all directed towards greater ,: compactness, generality, uniformity and simplicity (see [12]) ; in the first widely used implementation, APL\360, the main innovation was a uniform treatment of arrays, in the recently introduced APL\SV there is also an explicitation and systematisation of some of the "pragmatic" part of the language, via the concept of shared variables.

It must always be reme bered that APL was at the very begining a notation system urging for a rationalisation of mathematical - especially algebraic- notation. No spécial application was foreseen no special implementation recommanded, not even a parsing strategy.

Therefore APL's efficiency as a programming language is already in itself an achievement.

Yet, the user's opinion is that there is at least two facilities still missing for APL to be a most powerful and completely universal language :

- the possibility to handle data structures of a more general kind than arrays, such as trees, files and lists.

- better control structures.

In addition, the need is often felt to get faster execution of APL, which remains a remote prospect with a system completely interpreted at high level as is the present one.

So a new step forward is needed. But the constraints to be obeyed are very strong: keep the original spirit of the system, avoid proliferation of dialects etc...

On the other hand recent advances in programming language developement may be reinterpreted in the APL way and suggest interesting novelties.

This is the case, in particular of many works devoted to "extensible languages" research : (see [13], [16], [17], [18] , [21] .). A common feature of most of these tentatives is the explicit definition and manipulation of widely differing data structures via the notion of type (called sometimes mode or structure).

The possibility of going further with APL has been evoked in [20]. A specific proposal will be offered in [26].

The present paper is an intermediary step.

It has been found that the introduction in APL of the notions of general arrays (a slight modification of the concept introduced in [14] refered as G & M paper in what follows).

) and of types (in actual fact, a generalisation of this notion which we will call also "information predicates") answers the first of the above expressed needs : powerful facilities for data structures manipulation (with immediate applications to data-base management, symbolic and algebraic manipulation,...).

Furthermore the scheme here adopted provides the possibility of partial compilation of APL programs depending on the amount of information given by the user, via information predicates, in the program.
It is probable that compilation of the most often executed lines of a program would solve the problem of execution speed (see [15] for reference to previous tentatives of compilation of APL).

2. EXTENSIONS TO ARRAYS OF ARRAYS.

2.1. Any formal system is faced - at the very beginning-with the problem of data types and structures.

Let us .study the second one.

Iverson's book already contains a distinction of four kinds of structures : scalars,vectors, matrices and trees.

The possibility of working in algebras whose arguments could be non trivial data structures is a major motivation of APL , as K.IVERSON points out, page 2 of his book.:

" For example, separate and conflicting notations have been developed for the treatment of sets, logical variables, vectors, matrices, and trees, all of which may, in the broad universe of discourse of data processing, occur in a single algorithm. "

At this stage the general notion of an array is not explicited : only scalars, matrices and trees are specified. Arrays are only suggested in the book, §1.20 ("Levels of structure") ([1] ,p.39). Particularly interesting is the footnote *.

" * Further levels can, of course, be handled by considering a family of matrices ${}^{1}M, {}^{2}M, \ldots, {}^{n}M$, or families of families, ${}^{j}{}^{i}M$. "

The 1963 presentation of the language [2] and the 1964 joint work with FALKOFF & SASSENGUTH [3] forget about trees. Arrays of rank higher than 3 are introduced sometime beetween 1964 and 1966.

The next step is the celebrated "March on Armonk" [4] where S. FALKOFF submitted to strong APL users pressure admits that" related to the file handling and I/O question, is the generalization from arrays of scalars or single elements to arrays of arrays." ([4]) p.60)

Two years later we have the first concrete proposal, made by J. RYAN during the third APL users conference ([5]) p.8);a proposal is put up by J. BROWN in his thesis [6].

In 1972 a proposal was made by G. MARTIN and discussed informally at the APL/SEAS working committee. And finally in APL 73 three papers by EDWARDS, [7] MURRAY [8] and VASSEUR [9] . Finally comes the GANDHOUR and MEZEI paper [14].

2.2. Before going into the various proposals it seems appropriate to comment on the apparent slowness of all this process. And this can be done only by insisting on some peculiarities involved with any significant APL extension.

We restrict ourselves to APL/360 and to constants, that is ordinary (numerical or character) constants, and primitive functions. Then we note the following :

a) you can enter scalar or vector constants, not arrays of rank > 2 .

b) So you have to use "constructors" which are the primitive functions ρ , (outer products also delivers higher rank results). To these functions are associated inverses which give back the structure itself (here the monadic ρ).

c) admission of arrays as arguments for primitive functions cause no problem for the socalled "scalar functions", For mixed function extension is not straightforward..

Now it should be clear, that any extension of data structures raises the following problems.

a) defining "constructions" that would build the new data structures from keyboard admissible constants (i.e. scalars and matrices) and the corresponding "inverses" .

b) extending the meaning of scalar and mixed functions in such a way that nothing wrong happens when the new structure degenerates to an "ordinary" array.

From these requirements follows that a data structure extension implies in

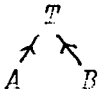
fact a complete reappraisal of most of the language and then necessarily interact with consideration coming for other modifications possibly under consideration;of course you may choose to improve those but then don't hope too much for an implementation !

2.3. The main published proposals which imply a significant extension of the language.

a) RYAN's proposal : [5]

construction of lists is achieved by a combination of semicolons and ( ) :

$T\leftarrow(A) \leftrightarrow$ 

$T\leftarrow(A;B) \leftrightarrow$ 

Selection is achieved by indexing along a path :

$T\leftarrow(A;((B;C;D);E;F))$

$B \leftrightarrow T[2\ 1\ 1]$

$E \leftrightarrow T[2\ 2]$

Measurement comes for a new primitive $\ddot{\rho}$

which returns a vector of lists :

$2 \leftrightarrow \ddot{\rho}[1]\ T$

$(0;3) \leftrightarrow \ddot{\rho}[2]\ T$

$(0;((0;0;0);0;0)) \leftrightarrow \ddot{\rho}[4]\ T$

Some others primitives are added, in particular

for catenation of lists

b) EDWARD's proposal [7] uses only one new primitive :

$$T \leftarrow \subset A \quad \leftrightarrow \quad \overset{T}{\downarrow} A.$$

which gives you vectors of anything.

Ordinary APL arrays are but special case(scalar arrays) of general arrays (having "relative scalars" which could be themselves arrays).

A special type of indexing is then required, where

Z[2] is different of Z[,2]

c) MURRAY's proposal [8]

Three primitivesare offered :

⊃ and ⊂ (conceal and reveal)

for construction and selection

⍺ for measurement.

d) VASSEUR's proposal [9]

Here construction of lists is offered through special brackets ({ }).

Measurement has three primitives : ⍺̈ Ḧ ⍴̈

Indexing is achieved via another bracketing system (⍧).

The APL 73 Congress in Copenhaguen was an opportunity to discuss all these proposals. A special session was devoted to this problem, chaired by P. BRAFFORT, with an active participation of D. FALKOFF and K. IVERSON. The APL fathers urged the audience to make a complete examination of GHANDOUR & MEZEI paper (not available at that moment) before going into new experiments.

2.5. GHANDOUR and MEZEI's proposal is by far the more extensive. Not less than 15 new primitive functions and operators are proposed. A clear distinction is offered between the concepts of a <u>function</u> and that of an <u>operator</u>. We shall retain many of their ideas but first criticize some aspects of their work.

The first objection one should make to G & M is that it contains no clear definition of what really are general arrays. This lack of an adequate formalisation entails a lot of semantic difficulties, together with the fact that the G & M conception of general arrays singularizes some objects they call "scalars" (which for them have the property that the enclose of a scalar is itself), creating some awkward effects ; for

instance, if the Ith element of a vector A is some array B, A∘I gives as result B if it is a scalar, and else it gives ⊃B ; the same happens in many other situations.

Another defect shows in a series of difficulties in the manipulation of general arrays due to a lack of means to examine the structure of an array : it is difficult to determine if a variable holds an ordinary or a general array. These difficulties in asserting the nature of objects can also be found in present APL, and can be traced down to the lack of a notion of "type". For example, in present APL and APL SV, `1↑'',⍳0` gives a blank as result and `1↑(⍳0),"` gives a zero while they seem to be the same object. This is due to the hidden notion of number or character type of an object and to the (never explicitely given) laws ruling the composition of these types under the primitive functions. In the given example, the rules used are :

- a character raveled to a number is a character
- and : a number raveled to a character is a number.

And there are two different empty objects : `''` which is a character and `⍳0` which is a number. We have other laws ruling operations on them, such as : `1↑1↓'A'` is a blank, and `1↑1↓1` is 0 which means that `1↓'A'` is `''` but `1↓1` is `⍳0` ...

Similar problems occur in unexpected conversions between reals,

integers and booleans, which sometimes cause a workspace to overflow when a boolean matrix is inadvertently converted to real (which happens after a division, even by 1!).

A last flaw we would like to point at in the G and M paper is the lack of proper examination of the syntactic difficulties arising from the definitions of functionals : they are an important notion discussed in G and M under the name of operators (we prefer to call them functionals, in accordance with common use in analysis and logic).

Some syntactic inconsistencies involving functionals are :

I:/+*A* is ambiguous : it is either *I*: /+ *A*
label plus-reduction

or *I*:/ + *A*
reduction along I^{th} axis

1.+2 is ambiguous : it is either 1. + 2
real number plus integer number

or 1 . + 2
integer external product with+ integer

3⍤×*B* is semantically ambiguous : its definition depends on the meaning of " x " used here, monadic or dyadic, and this cannot be determined from the context.

Such syntactic problems hamper further research : it would be useful to allow user-defined functionals in APL : this is impossible in

Ghandour and Mezei's scheme, since all the existing functionals have different syntaxes, and these are already inconsistent as has been shown. Introduction of any new operators cannot but increase the number of such inconsistencies.

3 - Some basic ideas for our APL extension

Here we list a series of concepts basic to describe our proposed extension. To each object is associated a set of attributes which we describe now.

A. Names

The basic objects of the language have a value (their internal representation)associated with a name (their external representation). We divide them in two classes according to their names :

1 - Autonymous objects, that is objects whose name coincide with the value, i.e. constants. They are :

-numbers, with syntax : [¯] integer [. integer] [E [¯] integer]

or a sequence of such separated by blanks ; here square brackets denote an optional item, and integer a non-empty sequence of digits. It is the present APL syntax, excepted that if a "." is present it must be preceded and followed by at least one digit (which may be zero). This is to prevent ambiguities with the functionals external and internal products.

- literals, wich obey APL syntax.

2 - Heteronymous objects, whose names are identifiers. The rules of formation of identifiers are :

α) a letter followed by a sequence of letters or digits (in letters we include the alphabet, the underscored alphabet, and Δ and Δ).

β) ⎕ followed by a sequence of letters or digits (eventually empty).

γ) ⍞

δ) A single special character (that is, any character not a letter, or digit, or one of ()':[]⍺).

This is a restriction compared to APL SV in a single respect : these rules exclude the identifier of the external product which is a sequence of two special characters ∘. . In APL-GA it is represented by the period alone.

The association of names with objects is less rigid than in APL SV. For instance, the objects with an identifier of classe δ) cannot be user-defined in APL SV but could be in APL-GA.

To enhance such a freedom, we would have liked the possibility of creating an infinity (or at least a number adding to 256 with the present number) of new characters. This is alas not possible within the present implementation (though the alpha fonts had been initially designed to be multiplied via overstriking with : underscore - overbar diaeresis "and quad (see [12])).

So the characters we will use here to represent new objects are not always the result of deliberate choise, but rather of the restrictions of the present I/O implementation, and so do not represent definitive options.

As a matter of fact a very slight modification in the interpreter solvesthis problem.

B. Order

Another feature associated with objects of the language is what we call "order"". Its definition is as follows :

- Objects which take no arguments are of order 0 (i.e. constants, ordinary variables, niladic functions and information predicates).

- Objects which take arguments of order 0 are of order 1 (i.e. monadic and dyadic functions).

- Objects which take arguments of order 0 and 1, at least one of them of order 1, are of order 2 (i.e. functionals).

- The operator axis of Ghandour and Mezei which can take a functional as argument, is of order 3.

This distinction provides a convenient framework for syntactic

analysis : approximately, objects, with order i have syntactic priority over objects of order less than i . Ideally, this would be not approximately but exactly true. We can specify here our criticism of G and M by saying that it is probably the fact that this is not true for functionals which prohibits the possibility of user-defined functionals.

C - General arrays

To sum up the structure of objects in our language, we can say that they have a name and a value ; with the value is associated an order. We now divide objects in four classes : functions, functionals, types and general arrays. We will discuss later the nature of functions, functionals and types. We will now define recursively what general arrays are.

A general array is either :

- a bit , that is one of two basic objects we denote 0 and 1 .
- or an object which we represent as :

$< n_1 , \ldots, n_k / a_1 , \ldots, a_N >$ where $a_i \in$ general arrays and where n_i, k are positive or null natural integers and $N = \prod_1^k n_i$ (if $k = 0$ then $N = 1$).

By this notation we mean that a general array is given by :

- a sequence of integers $n_1 , \ldots, n_k$ which we call the "rho" or the dimensions of the array ; k is the rank of the array.
- a list of general arrays a_i , in number the product of the dimensions.

We did not give other objects than the bit to build general arrays from, because all the usual objects (characters, real numbers, ...) will be built as special cases of general arrays built with bits.

This decomposition is not necessary but has two advantages :

- it allows within the language access to the bit representation of objects.
- it allows the language to be defined by the means of a small core and of a "standard prelude" (in the terminology of [13]) of definitions within the language, which makes the language much more tractable for semantic analysis and debugging (see the discussion of this point in [21]), which is a very desirable feature for an extensible language.

D. Types

What makes possible to build easily general arrays hierarchically from bits is the notion of "types", or information predicates An information predicate is essentially a predicate describing properties of a general array. To allow efficient manipulation and use of them, we had to restrict the expressible properties and the predicates we build as follows :

- there is the basic predicate "bit" describing an object which is a "bit".
- when we have two predicates p_1 and p_2, we can build a predicate $p_1 \vee p_2$ which describes any object which verifies one of the predicates p_1 and p_2.

- when we have a predicate p we can build the predicates :

$\supset p$ and $R \supset P$ if R is a sequence of integers meaning that the object refers to objects verifying the predicate p (that is, the object is of the form $< n_1 , \ldots , n_k / a_1 , \ldots , a_N >$ where a_i's verify p) and in the second case that in addition the object has R as dimensions (i.e. $R = n_1 , \ldots , n_k$) .

For type values (we say type or information predicate indifferently because our notion seems a natural extension of the current notion of type) the interaction between variables and constants is quite peculiar : we foresee two kinds of implementations :

- Implementations where the language is completely interpreted ; in these implementations the type values can be attributed to variables and them manipulated without any restriction.

- Implementations where the language is at least partially compiled ; therefrom assignnment of a type value to a new variable will be possible only by system command and can force to recompile code using another meaning for this variable .(*)

(*) This restriction is necessary because it is well known that the efficiency of a compilation depends largely on the amount of compile-time type checking which can be done.

In the interval between two redefinitions of a name holding a type value, this name will be considered as a type constant and the expression it was assigned to ; e.g. if INT was assigned the type $16 \supset BIT$, the operator $INT = 16 \supset BIT$ will give as answer false ; we will introduce a new operator $\Leftrightarrow$ "conforms to" and this time $(\Leftrightarrow INT) = 16 \supset BIT)$ will be true. This operator is useful in type expressions, enabling to consider atomic types as abbreviations for their definition.

We will come again to the distinction between atomic and other types and to their relation with the definition of general arrays in paragraph IV)

E - Left-values

The intermediate results of the computation of expressions are carried in objects whose names are not accessible to the user. We call them internal names ; among such names, we distinguish a class possessing what we call left-values (following the terminology of [21]). This means that the expressions they represent can be computed as a set of memory locations belonging to already existing variables. We call them left-values because they are exactly the permissible domain for the left-hand side of the assignement operator ($\leftarrow$) .

They are build according to the following rules :

- single names have left values

- the result of application of a selection operator to a left value is a left value. We shall see the definition of selection operators in part IV) ; They are indexation, compression, and take and drop for some values of the operands.

These left-valued expressions are the only instance of reference by name in APL ; all other references to objects are by value.

F - Extension mechanisms

The extensibility of APL-GA derives essentially from the concept of types. The main procedure used to build an extension is to introduce a new type of objects and extend part or all of the primitives of the language to allow interpretation of this new type.

With this purpose in mind, we first name the predicate value which will be the new type, assigning it to a new variable by the system command used to this purpose. For example : in polish style)*ISTYPE COMPLEX 2⊃FLOAT ∨INT*
or, in APL SV style : *COMPLEX'⎕ISTYPE 2⊃FLOAT∨INT*

After this, we now redefine our language primitives as acting on objects of the new type and corresponding conversion routines. All this and more is done by using an important primitive operator which we call a "cast" (following the terminology of [13]) and write " : " . Its uses are the

following :

a) Definition of new functions meanings when operands are of the new type : permissible form for the function headers in APL-GA is, therefore ;

$\nabla V3 \leftarrow [T1:]V1\ F\ [T2:]V2$

where : - items within brackets are optional : when omitted we have the usual APL header ; $V1, V2, V3$ are variable names and F is any identifier, and $T1$ and $T2$ are types.

The meaning is that following the header is the definition of the function F when acting on arguments of types $T1$ and $T2$ respectively. If we have an ordinary header, that is without type indication for the arguments, this means that the function is defined for all possible types. When we have written several such definitions of the same F , the rule is that the computer tries each one until it has found one which matches the arguments and then applies it. This allow to extend easily all primitives to new types.

b) transmission of informations to the compiler.

The only way to speed up the execution of a piece of text on a given machine is to increase the amount of compilation which is done on it, and for this we need to increase the amount of information given to the compiler : for instance, in present APL , when interpreting $A+B$, the main expense in time is due to calling a routine "+" fitted for the general case of A,B

arrays, one of which may be of bits and the other of reals (the + subroutine has more than 200 arguments in IBM's implementation !) when in fact we wanted to add integer scalars ; in the same way when interpreting $\rightarrow(A=0)/5$ we call two such general routines when we wanted to make a very simple conditional jump depending on a boolean scalar A .

The solution we offer here is to have an "interactive" compiler. That would be a compiler able to do a various amount of compilation depending on the amount of information given to it. (*)

In APL-G , we allow additional informations to be given to the compiler via the operator "cast". The general use is : $T:A$ which gives to the compiler the information that the object A conforms to the definition of the type T (this is close to use a)). If on execution the type does not match, an error message is produced. So, if we want to increase execution speed of code for A + B we can write $(I:A)+(I:B)$ and for $\rightarrow(A=0)/5$ we can write $\rightarrow((I:A)=0)/5$ if I is the type $(\iota 0)\supset INT$

(*) This is, perhaps, the fundamental point in compilation, since compilation is a translation preserving semantics, and the amount of semantic properties which can be proved on a program depends directly on the amount of information predicates given on each part of it (for this, see Floyd [22])).

These informations can also be given for the result of a function : a function header of the type $\nabla T:Z\leftarrow T1:X\ F\ T2:Y$ means that the result of F when applied to objects of type $T1$ and $T2$ will be of type T This allows the compiler to carry type informations through application of a function and thus enables the compiler to deduce informations for a whole program from initial assumptions.

A last point we have to make about compilation is that, of course, a piece of text involving the execute operator cannot be compiled except if there is no modification of the type of any variable during execution of the execute operator

4 - Description of the language APL-G

We will now give an organised description of the concepts and primitives of the language. This description will be quite brief for functions and functionals already present in APL SV or described in G and M and more extensive when our definition is new or differs singnificantly because of the use of concepts described above.

In what follows we discuss both implementations with and without compilation ; the only difference is some restrictions in the use of types in the second case. We also refer at several places to "our implementation", which is a simulation we made in APL SV to test our language and to prove the feasibility of simple and fast compilers for it.

Our description proceeds as follows : we first introduce the objects of the language and discuss which of them are primitive , then, after a brief review of the syntactic problems arising in our extension, we proceed to describe the primitives in order , beginning by data types and then functions and functionals.

A - Objects of the language.

We first summarize the characteristics of the four categories of objects of the language introduced above.

1 - Predicates or Types ; they refer to properties of general arrays We have the basic predicate bit , and the following operators acting on types

∨ dyadic $P1 \vee P2$ refers to an object verifying either predicate p1 or predicate p2.

⊃ monadic $\supset P$ refers to an object itself refering to objects verifying predicate p .

⊃ dyadic $R \supset P$ where R is an integer vector ; refers to an object which refers to objects verifying p and has R as dimensions.

As a special convention, when one of the elements of R is this means that the corresponding dimension is indeterminate.

Bit and all predicates designated by a single name are called atomic, in opposition to predicates designated by expressions involving ⊃ and ∨

One can create new atomic predicates apart form bit by :

-in interpreted implementations, assiging a predicate expression to a name,

e.g. : *INTEGER←16⊃BIT*

-in compiled implementations, by the system function *⎕ISTYPE* ,e.g. :

'INTEGER'⎕ISTYPE 16⊃BIT which cannot be used in definition mode and invalidates already compiled code using another meaning for *'INTEGER'*

We note here that we accept recursive definitions for types, for instance : *'BTREE'⎕ISTYPE 2⊃BTREE∨BIT* to define binary trees with bits as terminal elements.

We have one more operator on predicates apart form the cast (which was discussed in part II) :

monadic, argument should be atomic predicate p : refers to an object verifying the definition of p (which may or not have the type p).

2 - Arrays ; the definition we gave of them in last chapter was, in fact, only an approximation not taking into account precisely how atomic types different from bit can be attached to arrays. Their precise definition is : array = a bit or a couple $(< n_1 , \ldots , n_k / a_1 , \ldots , a_n >, p)$ where $n_i , k \in N , n = \prod_{i=1}^{k} n_i$, where the a_i's are arrays and p the predicate canonically attached to the array. We define recursively the predicate canonically attached to an array : - to a bit is attached BIT

- to an array $< n_1 , \ldots , n_k / a_1 , \ldots , a_n >$

such that the set of canonical types of the objects $a_1 , \ldots , a_n$ is $t_1 , \ldots , t_$

is attached the type $\supset t_1 \vee \ldots \vee t_n$.

This does not yet provides for the possibility that an atomic type different from BIT be attached to an array. Indeed, this is done with the operator cast written " : "

The value of p : A A array, p predicate atomic is A with type p if A conforms to the definition of p, otherwise it gives an error message.

3 - Functions

To summarize what we said in last chapter :

- the variances to standard APL are :

a) the general form of the header is ∇[[T1:]V1←][[T2:]V2] F [[T3:]V3] (items between brackets optional) which allows several definitions for the same function. The effect of a specification T1:V1 is exactly as if the last line of the function executed was V1←T1:V1

b) the identifier F can be any special character in addition to the ordinary names, which allows redefinition of primitive functions for new types.

4 - Functionals

They are . ⌿ and ⌸ . We will discuss them later.

B - The primitives.

1 - General discussion . From the extension mechanisms we gave, it can be easily seen that we could build our language by progressive extensions of a basic language with the only primitives :

- bit = {0,1} as data, bit as atomic type
- v , ~ (not) and "enclose" and "choose" as operators, and the functionals.

But in order to provide a language as pleasant as APL for the user, we have to give a lot of primitives in a "standard prelude" (following the terminology of [13]) available to all users.

The exact choice of primitives is largely a matter of taste, so the propositions we will put forth are more tentative than what preceded. The list of primitives we will give was build according to the following ideas :

- the standard APL functionals G and M ones really look fundamental, which implies their should be kept as they are. We follow the extension proposed by G and M which allows to apply functionals to user-defined functions including functions resulting from the application of functionals to other functions (this is natural for us since we do not make a rigid distinction between user-defined and other functions).
- the functions of standard APL are the result of a long experience and of a lot of thought, and so for most of them represent a natural choice, but : often we differ form G and M on the definition of their extension to general arrays ,

- the functions proposed in G and M are, we think, too numerous. We settle for a more restrictive set of basic functions.

2 - Syntactic problems

We exposed in part II) the syntactic problems arising from the definition of functionals in G and M . We will not try here to solve them in full generality (which is what we intend to do in another paper) but only to suppress the most striking inconsistencies :

- ambiguity of ⍁ : we solve it by suppressing the definition of ⍁ for dyadic functions, since the desired effect can be obtained with the functional ⍀ (scan).

- ambiguity of $I:⌿+A$: we suppress the axis operator written " : " allowing only the usual APL notation $⌿[I]+A$

- ambiguity of 1.+2 : the solution is that we prohibit the constant 1. and other constants ending in a period : the only justification for this was to specify that the constant is a real, and to do this it can be written 1.0 .

- another modification we suggest is the following :

In standard APL , the symbols / \ represent both a function and a functional. This introduces extended context dependance in syntactic analysis and in some cases ambiguities. This laxity in notation furthermore cannot be extended to other functionals and even prevents definition of new functionals

so we decided to suppress it.

/ and \ will stand for the functions compression and expansion.

⌿ and ⍀ will stand for the functionals reduction and scan.

and we will write /[1] , \[1] , ⌿[1] ,⍀[1] for what was written ⌿ and ⍀ in present APL . We found that this simplified and speeded up a lot syntactic analysis, and hence execution.

3 - The primitive data types

The usual data types are defined by the following atomic types canonically attached to them :

Character :	*C*	8⊃*BIT*
Integer :	*I*	16⊃*BIT*
Reals :	*F*	64⊃*BIT*

Of course these lengths are depending upon implementation.

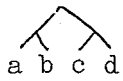
As a test of the ease of introduction of new data types in APL/G , in our present implementation we have two new types :

Complex numbers :	*CP*	2⊃*F*
One-variable polynomials :	*P*	⍬⊃*F*

Here we make some remarks about the input and ouput representation of constants and arrays.

We want first to make the following point about Ghandour's and Mezei

convention for general arrays, which uses an underscore : with this convention, there how exists 4 different kind of separators which can be used to represent objects which have a tree structure in APL :

1 - Parentheses and brackets are used in pairs of two and a tree of structure 

would be represented by *((AB)(CD))* using them.

2 - Quotes can represent a text within a text by doubling them ; the tree used as example above can be represented *'''AB''''''CD'''* .

3 - And, finally, we can uses underscores with the convention of G and M . the same tree is represented *AB_CD* .

So we have 3 completely different rules to represent trees with these different kinds of separators. I think that to simplify this is worth of study.

We do not propose any definitive solution for input and output of constants, having not done enough investigation on the subject : we follow standard APL or ad hoc conventions (e.g. [23] for complex numbers) .

4 - Primitive Scalar functions.

These are the functions which are primarily defined for basic scalars and extended to general arrays by a convention easily described with the "itemwise" functional (see [G and M] and definition below).

By "basic" scalars, we mean here an array of atomic type bit, *C*, *I* or, (and we include *CP* and *P* , too, in our implementation).

We note here that we allow manipulation of such objects in APL/GA when in present APL we can only manipulate objects of type ⍬bit , ⍬I ,⍬C or ⍬F . This simplifies writing of new extensions (see example below).

The list of primitive scalar functions is :

+ - × ÷ * ⍟ ⌊ ⌈ | ~ ∧ ∨ ⍲ ⍱ < ≤ = ≥ > ≠ ! ?

We follow present APL definition for these functions excepted for the following points :

1 - type of the result of these operators is not presently explicitely defined in the language, but can be determined by side effects in the present implementations, and one finds for example that :

- the result of a division or an exponentiation is always real, or the result of an addition or subtraction is never boolean, etc ...

In our implementation we instead choosed to always give to the result the lowest possible type on the hierarchy : bit-integer-real-complex.

For instance, `1+0` is a bit and `2*2` an integer.

2 - we extend many operators to complex numbers and polynomials.

For example | is the norm and × the unitary number of same argument as monadic functions on complex numbers Modulo is extended according suggesti of [23] for complex numbers and according to euclidean division for polynomials ; the real and imaginary parts of a complex A are naturally given by `A∘1` and `A∘2` (see "slice" and "choose" functions).

5 - Other primitive functions and functionals

Rho written ρ

monadic : for an array $\langle n_1 ,..., n_k / a_1 ,..., a_N \rangle$ gives the integer vector $\langle k / n_1 ,..., n_k \rangle$

dyadic : for AρB , A integer vector $\langle k / n_1 ,..., n_k \rangle$, B array gives an array $\langle n_1 ,..., n_k / a_1 ,..., a_N \rangle$ where $a_1 ,..., a_N$ is the list of the N first items of array B (cyclically repeated if there are not enough of them).

Note that we do not follow the definition of 0ρB in G and M finding it inconsistant.

Choose written ∘

We make first a few definitions :

- an index to an array $A = \langle n_1 ,..., n_k / a_1 ,..., a_N \rangle$ is an array $\langle k / b_1 ,..., b_K \rangle$ where $1 \leqslant b_i \leqslant n_i$. It indexes the element a_j of A where

$$j = b_k + (b_{k-1}-1)(n_k + (b_{k-2}-1)(n_{k-1} + \ldots \ (b_1 - 1)n_2)) \ldots) \ .$$

A path for an array $A = \langle n_1 ,..., n_k / a_1 ,..., a_N \rangle$ is defined recursively. It is a vector $P = \langle \ell / m_1 ,..., m_\ell \rangle$ where :

1 - $\ell > k$

2 - $m_1 ,..., m_k$ is an index to an item a_j of A

3 - either $K = 1$ (and we say P is a path to a_j) or $m_{k+1}, \ldots, m_1$ is a path for the array a_j .

We now define $A \circ B$ where A is an integer array and B an array :

- if $A = \langle n_1, \ldots, n_k / a_1, \ldots, a_N \rangle$, the result Z has as dimensions $n_1, \ldots, n_{k-1}$.

- each vector of A obtained by fixing the $k - 1$ first coordinates and varying the last is a path for B ; and if we fix the $k - 1$ first coordinate of A to $m_1, \ldots, m_{k-1}$ and obtain so a vector V , the element of index $m_1, \ldots, m_{k-1}$ of Z will be the element of B to which V is a path.

monadic $\circ A$ is a if $A = \langle /a \rangle$ and gives otherwise an error message.

Slice written $A[B]$ or $A \phi B$ indifferently, where B is a vector of integer arrays : we follow the G and M definition.

Itemwise functional : we follow the G and M definition ; it allows to define recursively the extension os scalar functions to arrays. If δ is a scalar function, we define its extension $\square EXT\ \delta$ as :

"$\square EXT\ \delta$ if arguments are not scalar, δ otherwise.

$\square EXT$ is implemented as a functional of the language accepting scalar functions headers as arguments. This allows very easy extensions of scalar functions to new types.

For example we give below a few lines of text in APL/GA : a definition of complex numbers and of some primitive functions on them :

```
      CP'⎕ISTYPE 2⊃E

∇CP:Z←CP:X+CP:Y
 [1]Z←X+Y        ∇

∇CP:Z←CP:X×CP:Y
 [1]Z←,X.×Y
 [2]Z←(Z[1]-Z[4]),Z[2]+Z[3] ∇

∇CP:Z←⍳ E:X
 [1]Z←0,X      ∇
```

and so one ... After this we have only to execute ⎕EXT'+',⎕EXT'×' , etc ... to have a full extension of complex number operators to any array containing complex numbers.

For the functions /\↑↓⌽⍉ and for the functionals ⌿⍀ and . we take the same straightforward extension to general arrays as G and M . But we give a meaning different from standard one to the scan operator acting on ordinary arrays, before extending it to general arrays. We first noted that for a vector of n elements and a non-associative function, calculation of scan takes $n(n-1)/2$ applications of the functions in contrast to $n-1$ for an associative function. If one takes as definition for $\delta\backslash V$ where δ is a dyadic function and V a vector $a_1, \ldots, a_n$, the vector :

$$a_1$$

$$a_1 \delta a_2$$

$$(a_1 \delta a_2) \delta a_3$$

$$\ldots\ldots\ldots\ldots$$

$$((.(a_1 \delta a_2)..) \delta a_n$$

Then the definition is identical with the present one for associative functions and takes only $n - 1$ operations for non-associative functions.

We them tried to compare the respective merits of these definitions : it seems that the only useful scans of non-associative functions are, with the present definition, $\neq\backslash$ and $\neq\backslash$ (they are the only non-associative scans whose result can be given any reasonable description).

They give respectively, for bit vectors : 1 at the first 1,0 elsewhere and 0 at the first 0,1 elsewhere.

With our definition we have the useful scans :

$\neq\backslash$: last 1 of each sequence of 1

$\neq\backslash$: first 1 of each sequence of 1

and furthermore, the present meaning of $\neq\backslash$ is not lost and can be obtained as $\neq\backslash\vee\neq$, with less applications of functions whenever $n > 5$!

Identical with " $\equiv$ "

As in [G and M] we define a boolean function which tests complete equality of two arrays.

Iota " ι "

Monadic iota is the unique array such that, for any array A, $A \equiv A \circ \iota \rho A$

it is defined only for integer vector arguments.

For dyadic iota, $\in$ and the functional $\boxtimes$, we take the same definition as [G and M].

Ravel : " , "

Monadic if $a = \langle n_1, \ldots, n_k / a_1, \ldots, a_N \rangle$ then, A is $\langle \prod_1^k n_i / a_1, \ldots, a_N \rangle$.

Dyadic : We found that in order, to build character arrays or other arrays to be used for outputing tables or other complicated formatted output, it would be extremely useful to have automatical extension for ravelling of some arrays. So the following extension fo "ravel" seems interesting :

1 - A,B is identical with $A,[(\rho\rho A)\lceil\rho\rho B]B$

2 - $A,[I]B$: it is authorized that $(\rho\rho A)\neq\rho\rho B$ only if $2\geq(\rho\rho A)\lceil\rho\rho B$ Then the array of lowest $\rho\rho$ has its ρ extended by a one in position i , and eventually another one if needed.

3 - If $(\rho\rho A)=\rho\rho B$ then $A,[I]B$ is identical with $(S\uparrow A),[I]T\uparrow B$ where $S=T=(\rho A)\lceil\rho B$ excepted at position i where $S[I]=(\rho A)[I]$ and $T[I]=(\rho B)[I]$

4 - If ρA and ρB differ only at position i we take the usual APL definition.

Format ⍕

For the same reasons as our extension of ravel, we modify the present APL/SV meaning of format so that ⍕ gives always exactly the image of the object on the paper, so that the $\rho\rho$ of the result never exceeds two.

Goto We found that the present control structure of APL is enriched and that many conditional branchings can be done faster with an extension

of goto to a dyadic meaning :

A→ B is allowed for B any boolean expression and means :

"if B then execute A" . The most interesting characteristic is that A is not executed if B is false, preventing an error message if execution is then impossible.

We can easily with this extension write some usual control expressions :

" if B then A else C" [1]A→B
[2]C→~B

"while B do A " →□LC[1],0ρA→B

Compare with standard APL traduction.

Finally we would like to indicate some modifications we suggest the definition of execute, □CR ,▼ and □FX .

They are as follows :

1) "Execute" accepts a two-dimensional array as argument, executing sequentially each line.

2) ▼ is defined for functions, giving as result the text less the header.

3) While □CR gives only the header, and for an ordinary variable gives as result its name (the argument is no more to be put within quotes).

4) and □FX is dyadic, needing as left argument a function header, and as right argument a function text less the header.

The role of these modifications is essentially to make more logical and

and uniform the diverse relation existing between these functions.

Further, the modification of $\Box FX$ is very useful when defining different but very similar versions of a function for different types.

V - Implementation

Our implementation is - presently - an APLSV simulation to test the feasibility of a compiler for our language. A very efficient compiler (that is, taking is account the double level of interpretation) accepting almost our whole language could thus be made, along the following principles.

A) Memory allocation.

Memory allocation and garbage collection are always, effected through reference counting (cf. [19] , chap. II). This is possible since, due to the concept of left-values, all data structures, including data refering to other data, can be described as trees. It allows efficient garbage collection and keeps duplication of identical data to a minimum (insteed of copying some data,we increase its reference count by one).

B) Parsing.

Our concept of order and the restrictions we discussed under C) of Chap. IV allows us to parse a line with a (1,1)BRC syntax [24] , with a small table. This is the main reason why our compiler is fast and needs a

modest amount of memory.

C) APL SV representations.

They are as follows :

memory blocks and all auxiliary functions and variables for the compiler are represented by APL SV variables beginning by Δ or $\underline{\Delta}$: this is the only class of names legal in APL/GA but illegal in our implementation. Memory blocks are represented by variable Δnnn where nnn is an integer. Reference count is done by the means of an integer vector ΔNAMES whose i^{th} element is the reference count of Δi

An APL-GA general array is represented by an APL SV vector having the same name, and whose structure is as follows (much like standard APL M- entry [25]).

- The first element is the type, encoded (via a symbol table) as a positive integer . the same integer is taken negative if the array is a left-value.
- The second element is the rank of the array.
- The following elements are the dimensions.
- after this is the list of elements. If these are not bits, they are then themselves arrays. They are then represented by the index of the memory block where they are stored.

D) System and compilations

The present implementation includes :

- a compiler $\Delta COMP$ which can compile a single line of APL test

- a function compiler $\Delta FCOMP$ which can compile any APL function and which uses $\Delta COMP$.

- and a superviser ΔSP which can accept lines and execute them on the spot. It is very short and we give it here :

```
     ∇ SP AA;LΔ
[1]    ⍞←6↑''
[2]    →(0=ρLΔ←(¯1+(LΔ≠' ')ι1)↓LΔ←⍞)/1
[3]    ⍎⎕FX((ρLΔ)[2]↑'Δ'),[1] LΔ←ΔCOMP LΔ
[4]    →AΔ/1
[5]    CTIME
[6]    →1
     ∇
```

Here ΔP is a routine implementing our proposed extension for ravel and *CTIME* gives a timing indication.

The programs comprising our compiler can be divided in two parts :

- the compiler itself

- semantic routines implementing primitive operations. Our APL SV representation allows these semantic routines to be APL functions acting directly on our APL/GA objects. We will now describe the core of our compiler, which consists of about 100 lines of APL. The main function is

which accepts as input a line of APL|GA and gives us output an equivalent text in APL|SV . It uses as auxiliary functions :

$\Delta SCAN$: a function of 2 character vector arguments returning us result the index of the first occurence of the first argument in the second.

ΔNEWN : a function returning the name of a new block of memory (that is, Δi where i is the first index such that the reference count of Δi is zero).

ΔCODE : a function returning the internal representation of its argument

ΔADIC : a function returning the name of the semantic routine implementing the APL/GA function represented by the second argument of *ΔADIC* (when it has an adicity equal to the first argument of *ΔADIC*).

ΔN, *ΔO*, *ΔP*, *ΔS* are other functions discussed later.

The global variables used are :

ΔSPEC, *ΔDIG*, *ΔCHARS* : vectors of characters initialised by the function *ΔINIT* .

ΔFTABLE : character vector used as a symbol table for the functions recoynized by the system at a given time.

ΔCOD : dictionary used by *ΔADIC* to find the name of a semantic routine implementing some APL/GA primitive function.

ΔG : Parse table for the (1 - 1) BRC grammar.

In *ΔCOMP* , lines 1 - 15 accomplish the lexical analysis and the encoding of constants, and the other lines do the parsing and translation.

The local variable are :

T - the text, preceded and followed by the character Δ used as a "marker".

S - a vector of integers of same length as the text, used to store the results of lexical analysis : after it, all characters, of T belonging to the same

word correspond to positions of s containing the same integer, which is the position in T of the first character of that word..

N - a vector completing the information of S . N[i] contains an integer representing information on the nature of the word beginning at i^{th} position of T .There is also an encoding of this information as characters, with the following correspondance.

Integer encoding	character encoding	Nature of word
0	*C*	a constant
1	0	the name of an object of order 0
2	1	the name of an object of order 1
3	Δ	a marker of beginning or end of line
4	)	)
5	≠	the functional ≠ or ≤
6	.	the functional .
7	..	the functional ..
8	]	]
9	[	[
10	(	(
11	:	:
12	←	←
13	→	→

The lines 1 - 7 of build the vector S . Line 8 builds N but distinction between objects of order 0 or 1 is not yet done. It is done in lines 9-11 , by consulting *ΔFTABLE*

Lines 12-15 replace constants by their internal representation.

Parsing then proceeds according to the following sheme :

P is a vector used as pushdown store, where a word is represented as the index of its first letter in T . It is initialised to the list of word of the line and scanned right to left by the variable I . At each step, an action is taken taking only in account the nature of the words

$P[I-1], P[I], P[I+1]$.

These actions are of the kind : generate a line of text for an operator acting on one or two operands , and substitute in P the result for the operator and the operands.

The syntax is fully represented by the table ΔG , where $\Delta G[I1;I2;I$ is a letter representing the address of the action to be taken when $P[I-1]$ is of nature $I1$, $P[I]$ of nature $I2$ and $P[I+1]$ of nature $I3$.

Here is a list of all triplets

STATABLE

```
0000000000000000000011111111111ΔΔΔΔΔΔΔΔΔΔ))))))))))))))))≠≠≠...¨¨]]]]
11)))))."]]]]]]][::←→000001)."[000011Δ¨→0011)))))."]][←→1."[01.1.0000
0[1Δ).[1101Δ)]←00→001Δ)][001101Δ:←0[Δ10Δ)0[01Δ)111)0000110Δ01011Δ)]
DAKYJKKGGKKKKKKCKKBYJKKKJMKOHKJFVJMAWHUKKKKKKKKKKKKKKYRGGAKKGKGJKKK

]]]]]]]]]]]]]]]][[[[((((((((:::::←←←←←←←←→→→→→→
011)))))."]]]]]←→01¨00011."01¨→000001¨0001¨
[0[01Δ)1101)]←00]011)[0[11Δ0101Δ)][011Δ[01
JKAKKKKGKKKKKKKKYKMHJXJMAGHKMHKJKKKJMHJKJMH
```

$A\ B.\Delta S\ C$ substitutes in P words $P[I+A]$ to $P[I+B]$ by word C and ΔN serves to substitute the nature of c . ΔO i gives the i^{th} item of the line.

The meaning of the actions are :

A : Axis operator

B : ←

C : choose function

D : a dyadic function

E : error

F : end of parsing

G : monadic functional acting on dyadic function

H : itemwise of monadic functions

J : skip back one word

K : skip forward one word

M : monadic function

Q : external product

R : ⌿ or ⍀ functional

U : monadic →

V : label

Y : dyadic →

ΔP is a function which catenates its argument us a knew line to the object-program PG .

We give here, in addition to the listing of all these functions, the array ΔCOD and some examples of translations.

E - Examples and listings.

Here are some examples of output of Δcomp

```
      ΔCOMP'F1:Z←(N[P]=I←0)/⍳⍴P'
F1:Δ140←ΔRH P
Δ141←ΔIT Δ140
Δ142←'I'ΔIS 5 0 139
Δ143←'N'ΔCH P
Δ144←Δ143 ΔEQ Δ142
Δ145←Δ144 ΔCM Δ141
Δ146←'Z'ΔIS Δ145
```

Notice the treatment of axis operator

```
      ΔCOMP'A:→(⍲[1]+A≠7+X∈''ABCD'')/C←4'
A:Δ103←'C'ΔIS 5 0 102
Δ104←X ΔEL 5 1 4 98 99 100 101
Δ105←5 0 97 ΔAD Δ104
Δ106←A ΔNE Δ105
'⍲'ΔAX 5 0 96
'Δ107'ΔRD'ΔAD'
Δ108←Δ107 Δ106
Δ109←Δ108 ΔCM Δ103
→Δ109
```

We deal easily with deep parenthesing.

```
      ΔCOMP'(((A+B)×(L+((P+O)+U))))'
Δ147←P ΔAD O
Δ148←Δ147 ΔAD U
Δ149←L ΔAD Δ148
Δ150←A ΔAD B
Δ151←Δ150 ΔMP Δ149
ΔUT Δ151
```

Here is an example of treatment of dyadic go to

```
      ΔCOMP'K←L+1→B≠K←L+2→B≠1'
Δ113←B ΔNE 5 0 112
→(~Δ113)/7+⎕LC
Δ120←L ΔAD 5 1 3 117 118 119
Δ121←'K'ΔIS Δ120
Δ122←B ΔNE Δ121
→(~Δ122)/3+⎕LC
Δ129←L ΔAD 5 1 6 123 124 125 126 127 128
Δ130←'K'ΔIS Δ129
```

We now give the listing of Δ comp

```
    ∇ PG←ΔCOMP T;S;Z;N;P;I;L
[1]   PG←0 1 ρ''
[2]   S←(~S∨≠\S←T='''')×1+(T←'Δ',T,'Δ')∈ΔSPEC
[3]   S←T[P←((T≠' ')∧(S≠¯1ϕS)∨S=2)/ιρT]='.'
[4]   →('Δ'≠T[P[2]])/CT
[5]   ΔP ¯1↓(P[2]-1)↑T
[6]   →0
[7]  CT:S←P[+\(ιρT)∈P←(~(Z∧¯1ϕZ∨S)∨(S∧(1ϕZ)∧¯1ϕZ←T[P]∈ΔDIG))/P]
[8]   N←+/(ΔCHARSιT[S])∘.≥ΔCHARSι'A+Δ)≠.¨][(:←→'
[9]   →(0=ρZ←(N[P]=I←1)/P)/F1
[10] IT1:N[Z[I]]←1+(ρΔFTARLE)≥(' ',(Lι' ')↑L←ΔO PιZ[I]) ΔSCAN ΔFTABLE
[11]  →((ρZ)≥I←I+1)/IT1
[12] F1:Z←(N[P]=I←0)/ιρP
[13] IT2:→((ρZ)<I←I+1)/F2
[14]  P[Z[I]]←(⍕ΔCODE⍎ΔO Z[I]) ΔN 1
[15]  →IT2
[16] F2:I←''ρ¯1+ρP
[17] K:I←I-1
[18] IT:→⍎ΔG[N[P[I]];N[P[I+1]];N[P[I+2]]]
[19] A:→(N[P[I+5]]≠2)/3+⎕LC
[20]  I←I+4
[21]  →M
[22]  ΔP ''''',(ΔO I+1),'''ΔAX ',ΔO I+3
[23]  0 4 ΔS P[I+1]
[24]  →IT
[25] B:ΔP(L←ΔNEWN),'←''',(ΔO I),'''ΔIS ',ΔO I+2
[26]  ¯1 2 ΔS L ΔN 1
[27]  →K
[28] C:ΔP(L←ΔNEWN),'←''',(ΔO I),'''ΔCH ',ΔO I+2
[29]  ¯1 3 ΔS L ΔN N[P[I]]
[30]  →K
[31] D:ΔP(L←ΔNEWN),'←',(ΔO I),' ',(2 ΔADIC ΔO I+1),' ',ΔO I+2
[32]  ¯1 2 ΔS L ΔN 1
[33]  →K
[34] E:1↓(P[ρP]-1)↑T
[35]  ((0⌈P[I]-2)ρ' '),'∧SYNTAX ERROR'
[36]  →0,0ρDIAGNOSE
[37] F:→((ρPG)[2]≥'ΔIS' ΔSCAN(-ρPG)[2]↑,PG)/2+⎕LC
[38]  ΔP 'ΔUT ',ΔO I+1
[39] W:→0
[40] G:ΔP ''''',(L←ΔNEWN),'''',(1 ΔADIC ΔO I+1),'''',(2 ΔADIC ΔO I+2),''''
[41]  0 2 ΔS L ΔN 2
[42]  →IT
[43] H:ΔP ''''',(L←ΔNEWN),'''ΔIM''',(1 ΔADIC ΔO I+2),''''
[44]  0 2 ΔS L ΔN 2
[45]  →IT
```

```
[46] J:I←I+1
[47]  →IT
[48] M:ΔP(L←ΔNEWN),'←',(1 ΔADIC ΔO I+1),' ',ΔO I+2
[49]  0 2 ΔS L ΔN 1
[50]  →IT
[51] Q:ΔP ''''',(L←ΔNEWN),'''ΔEXT''',(2 ΔADIC ΔO I),' ',(2 ΔADIC ΔO I+2),''''
[52]  ¯1 2 ΔS L ΔN 2
[53]  →K
[54] R:ΔP ''''',(L←ΔNEWN),'''ΔRD''',(2 ΔADIC ΔO I+1),''''
[55]  ¯1 1 ΔS L ΔN 2
[56]  →K
[57] U:ΔP '→',ΔO I+2
[58]  0 2 ΔS P[I]
[59]  →IT
[60] V:PG←((1,ρZ)ρZ),[1]((1-1↑ρPG),(ρPG)[2]⌈ρZ←(ΔO I+1),':',PG[1;])↑PG
[61]  ¯1 2 ΔS P[I]
[62]  →IT
[63] X: ¯1 2 ΔS P[I+1]
[64]  →K
[65] Y;Z←ΔCOMP T[(+/S∈L)↑⍋(L←1↓I↑P)⍳S]
[66]  ΔP '→(~',(ΔO I+2),')/',(⍕1+(ρZ)[1]),'+⎕LC'
[67]  PG←(((ρPG)⌈0,L)↑PG),[1]((ρZ)⌈0,L←((ρPG)⌈ρZ)[2])↑Z
      ▽
```

And now a listing of the auxiliary functions used with it.

```
∇ ΔINIT
  ΔALP←'ABCDEFGHIJKLMNOPQRSTUVWXYZΔABCDEFGHIJKLMNOPQRSTUVWXYZ⍙⎕⍞'
  ΔDIG←'¯0123456789'
  ΔSPEC←'+-×÷*⌈⌊|∧∨≠>≥=≤<⍲⍱~⍴∊⍵?↑↓⍳○⍟∘⊂⊃∩∪⊥⊤;\/,⍺⍉⌽⊖⍋⍒⌹!⍕⍎I⍙)⌿⍀.¨][(:←→ _⍝∇⍫'
  ΔCHARS←'''',ΔDIG,ΔALP,ΔSPEC
  ΔNAMES←⍳0
  ΔFTABLE←' C P ',, 50 2 ↑ 50 1 ⍴ΔSPEC
∇


    ∇ Z←B ΔSCAN A
[1]   Z←(∧⌿(¯1+⍳⍴,B)⌽(,B)∘.=,A)⍳1
    ∇


    ∇ ΔP L;A
[1]   PG←(A↑PG),[1](A←(⍴PG)⌈0,⍴,L)[2]↑,L
    ∇


    ∇ Z←I ΔN A
[1]   S←S,(⍴I)⍴Z←1+⍴T
[2]   T←T,I
[3]   N←N,(⍴I)⍴A
    ∇


    ∇ A ΔS B
[1]   P←((I+A[1])↑P),B,(I+A[2])↓P
    ∇


    ∇ Z←ΔO I
[1]   Z←(S=P[I])/T
    ∇


    ∇ Z←ΔNEWN
[1]   →((⍴ΔNAMES)≥Z←ΔNAMES⍳0)/4
[2]   ΔNAMES←ΔNAMES,1
[3]   →5
[4]   ΔNAMES[Z]←1
[5]   ΔCΔ←ΔCΔ,Z
[6]   Z←'Δ',⍕Z
    ∇


    ∇ Z←I ΔADIC ITEM;L;K
[1]   →((⍴ΔSPEC)<K←ΔSPEC⍳L←1↑Z←ITEM)/0
[2]   Z←,ΔCOD[I;K;]
    ∇
```

Here is the dictionnary fo traduction of primitive.

What is output here is 1 32 ΦΔ COD

```
    CODES
+-×÷*⌈⌊|∧∨≠>≥=≤<⍲⍱~⍴∊⍵?↑↓⍳○⍟∘⊂⊃∩∪⊥⊤;\/,⍺⌽⍉⊖⍋⍒⍕!⍱⍎⌶
ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ ΔΔΔΔΔΔΔΔΔ ΔΔΔΔΔΔΔ ΔΔΔΔΔΔΔ
PMSIECIA             NR   H   IPED R D   E   R TPTOCMG
LNGVXLFB             TH   D   TIPC F C   N   V RH1DUIM

ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ ΔΔΔΔΔΔΔΔΔ  ΔΔΔΔΔΔ ΔΔΔΔΔΔΔ
ABMDPSIMEOHGGELLNH RE DTDPTLCIG  EDLECC RRR  SC
DBPVMPHDTRETEQETDR RL LKRSGGHSA  DDKPHT ST1  LB
```

To finish we give a glimpse of what is a session of APL G.

```
X←P̲ 1 1
0H0M0S680
X
 1+X
0H0M0S980
T←X×X
0H0M1S520
T
 1+2X+X2
0H0M1S820
⎕←U←T×T
 1+4X+6X2+4X3+X4
0H0M2S640
Y←C̲ 2 3
0H0M3S280
Y
 2+3ι
0H0M3S560
Y×Y
 -5+12ι
0H0M4S40
A←1
0H0M4S340
```

REFERENCES.

[1] K. IVERSON A programing language J. WILEY 1962.

[2] K. IVERSON Formalism in programming languages. CACM 7 1964 p.80
(Presented at a working conference on Mechanical languages structures Princeton N.J. August 1963).

[3] S. FALKOFF , K. IVERSON, E. SUSSENGUTH.
A formal description of system 360 IBM Sys j. 3 1964, 181.

[4] Proceeding APL users conference at S.U.N.Y. Binghampton 1969.

[5] A summary of the presentations at APL User conference Workshop 3 Quotquad 3 1971

[6] J. BROWN A generalization of APL Ph. D Syracuse University 1971

[7] E. EDWARDS Generalized arrays (lists) in APL in P. GJERLOV . H.S. HELMS , J. NIELSEN (ed. APL Congress 73 p.99

[8] R. MURRAY On tree structure extensions to the APL language id. p.333.

[9] J. VASSEUR Extension of APL operator to tree like data structures id. 457.

[11] K/SMILLIE APLISP : a simple list precessor in APL quotquad 3 1972

[12] Falkoff and Iverson " the design of APL " IBM J. Research July 1973

[13] Van Wijngaarden and al" report on the algorithmic language Algol 68". Mathematische centrum, Amsterdam, 2/1969.

[14] Ghandour and Mezei "general arrays, operators and functions "IBM J. Research July 1973.

[15] Van Medel Colloque APL IRIA 1971 p.339.

[16] Garwick, Jan V. "GPL, a truly general-purpose language" Comm. ACM vol 11 n°9 (9/1968) pp.634-38

[17] A survey of extensible languages SIGPLAN notices.

[18] Irons, E.T. "experience with an extensible language "Comm. ACM Vol.13 n°1 (1/1970) pp.31-40.

[19] Knuth, Donald E. " the art of computer programming", vol. 1-3 , Addison-Wesley, Massachusssts 1968.

[20] P. Braffort . Paper III-1 this issue.

[21] Ben Wegbreit "studies in extensible programming languages" report AD 71 5332 Harvard University, Cambridge, Massachussets.

[22] Floyd. R. Assigning meanings to programs "Proc. Symp. Appl. Math. Vol.19

[23] E.C. Mc Donnel, "complex floor "APL-congress 73 (North-Holland) p.229

[24] Aho, currents in the theory of computing.

[25] APL - 360 OS System Mannual LY 20-0 678-0.

[26] APLX - P. Braffort and J. Michel to appear.