Aspects de la réalisation d'un système APL optimisé

Christian Bertin

ECOLE NATIONALE
SUPERIEURE DES MINES
DE SAINT-ETIENNE

INSTITUT NATIONAL
POLYTECHNIQUE
DE GRENOBLE

N° d'ordre : 26 II

THESE

présentée par

Christian BERTIN

pour obtenir le grade de

DOCTEUR-INGENIEUR
"SYSTEMES ET RESEAUX INFORMATIQUES"

ASPECTS DE LA REALISATION D'UN SYSTEME APL OPTIMISE

SOUTENUE A SAINT-ETIENNE LE 1er DECEMBRE 1981 DEVANT LA COMMISSION D'EXAMEN :

M. L. BOLLIET — Président

Mme A. DECROIX
MM. J.J. GIRARDOT
S. GUIBOUD-RIBAUD
R. MAHL
M. NAKACHE — Examinateurs

ECOLE NATIONALE
SUPERIEURE DES MINES
DE SAINT-ETIENNE

INSTITUT NATIONAL
POLYTECHNIQUE
DE GRENOBLE

N° d'ordre : 26 II

THESE

présentée par

Christian BERTIN

pour obtenir le grade de

DOCTEUR-INGENIEUR
"SYSTEMES ET RESEAUX INFORMATIQUES"

ASPECTS DE LA REALISATION D'UN SYSTEME APL OPTIMISE

SOUTENUE A SAINT-ETIENNE LE 1er DECEMBRE 1981 DEVANT LA COMMISSION D'EXAMEN :

M.	L. BOLLIET	Président
Mme	A. DECROIX	Examinateurs
MM.	J.J. GIRARDOT	
	S. GUIBOUD-RIBAUD	
	R. MAHL	
	M. NAKACHE	

Avant tout, je tiens à remercier :

Monsieur Louis BOLLIET, Professeur à l'Institut de Mathématiques Appliquées de Grenoble qui m'a fait l'honneur de présider le jury de cette thèse.

Monsieur Serge GUIBOUD-RIBAUD, Ingénieur chez HEWLETT PACKARD, qui est à l'origine de ce travail et qui l'a continuellement guidé alors qu'il était Directeur du Département Informatique Appliquée de l'Ecole des Mines de Saint-Etienne.

Messieurs François MIREAUX et Jean-Jacques GIRARDOT, Ingénieurs de recherche à l'Ecole des Mines de Saint-Etienne, sans la longue expérience et l'infinie patience desquels un tel travail n'aurait pu aboutir.

Monsieur Robert MAHL, Chargé de mission à la D.G.R.S.T. et Monsieur Michel NAKACHE, Ingénieur commercial au département APL de la S.L.I.G.O.S., qui ont accepté de s'intéresser à ce travail en faisant partie du jury.

Madame Agnès DECROIX, Ingénieur chez PHILIPS, qui a assuré un suivi vigilant de l'évolution de ce travail et ne m'a jamais ménagé ses conseils.

Messieurs BROSSARD, VELAY, DARLE et LOUBET qui ont assuré avec beaucoup de soins et de gentillesse le tirage de cet ouvrage.

o

o o

o

RECAPITULATIF

INTRODUCTION

I - AVANT-PROPOS

K.E. Iverson a défini le langage APL en 1962 dans un ouvrage intitulé "A Programming Language". Malgré les difficultés nouvelles que posait ce langage aux implémenteurs, des interprètes APL furent rapidement disponibles chez I.B.M., d'abord à titre expérimental en 1965, puis commercialisés sur ordinateurs 1130 et 360 (R3,R7). En effet, ce langage s'était d'emblée révélé intéressant pour toutes les applications réclamant plus de programmation que de longs calculs.

Depuis cette époque, d'autres interprètes APL ont vu le jour chez d'autres constructeurs (R1,R2,R5). Réalisés pour la plupart sur de gros ordinateurs, ils réclament des ressources hors de proportion avec les moyens habituels des ingénieurs ou des étudiants.

Parallèlement, de nombreuses recherches ont été menées pour tenter d'accélérer l'interprétation du langage. P.S. Abrams a, en premier, introduit certaines notions fondamentales d'optimisation et envisagé la construction d'un matériel muni d'un code machine très voisin du langage APL (I5). Plusieurs propositions tendant à restreindre le langage afin de pouvoir le compiler plus facilement ont été faites (I10,I13,I15,I16,I18). Chez Hewlett Packard, un système APL a été réalisé et doté d'un compilateur dynamique incrémental générant à partir du langage source un code directement exécutable et pouvant être réutilisé.

D'autres approches, moins théoriques, ont consisté à réaliser un maximum de fonctions possibles au niveau matériel, en microprogrammant un sous-interprète scalaire ou vectoriel et en rédigeant ensuite dans l'APL restreint ainsi obtenu l'interprète complet.

Cependant, curieusement, on constate que depuis 15 ans les principaux systèmes APL ne tournent que sur de gros ordinateurs, en utilisant presque toujours une technique d'interprétation naïve dont toutes les fonctions sont réalisées à un niveau purement logiciel. On peut évidemment se demander pourquoi ? Les justifications des concepteurs sont d'ordre divers :

a) - Réalisation d'un système APL complet, destiné à une exploitation effective : ces implantations ne font, en général, l'objet d'aucune innovation. Dans ce cas, le constructeur, désireux d'inscrire APL sur son catalogue, s'adresse pour sa réalisation à une société de service qui, à cause d'impératifs d'ordre économique, n'a ni le temps ni les moyens d'approfondir l'état de l'art. Ou bien le constructeur réalise lui-même son système APL, auquel cas des considérations de fiabilité et de rentabilité orientent son travail vers des recettes connues, éprouvées et sans risque, plutôt que vers des nouveautés aux résultats aléatoires.

b) - Application de nouveaux algorithmes, pour l'optimisation de l'interprétation ou la représentation des structures de données : l'expérience montre que de tels interprètes sont rarement utilisés, même lorsque la réalisation a été poussée jusqu'au stade où le produit est devenu opérationnel. La raison en est, le plus souvent, qu'un seul aspect du problème a été correctement abordé (celui qui relevait des nouveaux algorithmes), et que la réalisation pèche par d'autres côtés, ce qui interdit son exploitation.

c) - Résolution des problèmes inhérents au langage selon les méthodes habituelles, mais de nouveaux problèmes sont rencontrés dûs à des conditions inhabituelles de réalisation : par exemple, celle-ci a lieu sur un petit ordinateur, ou une gestion de fichiers est intégrée à l'interprète, ou encore de nouveaux opérateurs sont introduits dans le langage, etc.

Le travail que nous présentons se rattache aux deux dernières approches, notre but étant de réaliser, sur une petite machine :

- un système complet, fiable, sans réelles limitations :

L'expérience d'APL/16 montre qu'une réalisation imposant de trop fortes limitations (rang des objets limité à 4, capacité de l'espace actif limitée à 64 K-octets, utilisation d'une mémoire secondaire segmentée rendant la sauvegarde de grosses variables actives impossible) crée une gêne sérieuse chez de nombreux utilisateurs du système (R2). Disons immédiatement que seule la mise en oeuvre d'une mémoire virtuelle simulée permettra de lever efficacement l'ensemble de ces limitations.

- un réel système , avec de bonnes performances :

Il sera obtenu par l'emploi de techniques d'optimisation connues ou nouvelles (beating, j-vecteurs, partage de données, compilation des accès aux objets) proprement intégrées et conférant au système complet une qualité industrielle le plaçant en très bonne position dans la gamme des mini-ordinateurs.

- un système extensible et un support d'expérimentation :

Ce point contribue à l'originalité du travail puisque cet aspect de la réalisation est difficile à concilier avec les objectifs de qualité précédents. Il a été obtenu en laissant ouvertes, dès la conception, des voies aux extensions suivantes :

1°) Vers une utilisation multiconsole :

Bien que prévue au départ pour un seul utilisateur, la différentiation des fonctionnalités des deux processus constituant le système APL, le superviseur et l'interprète, a été conçue afin qu'une version multi-console puisse être déduite de la version mono-console au prix d'un faible investissement. Ce point précis a fait l'objet d'un travail antérieur à cette thèse (G5) et a permis de séparer très précisément les fonctions de l'un et de l'autre. L'univers de l'interprète est essentiellement la zone de travail active. Son principal rôle consiste à exécuter une ligne du langage. Accessoirement, l'interprète communique avec l'extérieur par l'intermédiaire du superviseur. Le superviseur dialogue avec l'utilisateur. Il assume les tâches ingrates (celles qui ne correspondent pas à ce qui est prévu dans le langage, mais qui doivent être cependant accomplies, dans la mesure où l'on travaille sur une machine réelle).

2°) Vers les tableaux généralisés :

Cette généralisation du langage fait actuellement l'objet de recherches très actives aux Etats-Unis et on peut s'attendre à ce qu'à nouveau, un jour ou l'autre, en l'absence de standard, I.B.M. impose sa seconde version d'APL comme une référence. Le système expérimental N.A.R.S.(Nested Arrays System) proposé par la S.T.S.C. est un bon modèle du genre (T12).

Afin de pouvoir réaliser une telle extension, la gestion de mémoire a été prévue pour gérer non pas de simples blocs mais des structures arborescentes. Indépendamment des tableaux généralisés, une telle gestion a permis de manipuler de manière homogène différentes structures de données mises en oeuvre dans le système APL/857 : tables des symboles, fonctions définies, pile des contextes, objets optimisés, fichiers, etc...

3°) Vers un système de fichiers ou de variables partagées :

Le superviseur a été prévu pour gérer simultanément différents types d'espaces virtuels. L'un est l'espace de programme partagé entre tous les utilisateurs. Le second représente l'espace de travail actif et contient les données propres à chaque utilisateur. Les troisième et quatrième peuvent être par exemple un espace partagé entre tous les utilisateurs, dans lequel on peut ranger les données partagées, et un espace fichier qui peut représenter un des fichiers accédés à un moment donné par l'utilisateur.

4°) Vers des moyens de protection des applications :

Grâce à l'introduction de nouvelles commandes système, on peut fournir une zone de travail qui ne comporte qu'une fonction principale accessible, qui peut manipuler des données masquées, dans laquelle par la définition de trappes, on peut spécifier des déroutements en cas d'erreur, etc.

Ce système a été effectivement réalisé sur un ordinateur Philips P857, à l'école des Mines de St-Etienne. Il a demandé sept hommes-années de développement, dont plus de la moitié à l'auteur. La description complète d'un tel travail n'entre pas dans le corps d'une thèse et n'y apporterait qu'un intérêt très relatif.

La thèse ne contient donc que certains des aspects les plus intéressants de la contribution de l'auteur, qui, de ce fait, ont pu

être exposés plus précisement.

II - PLAN DE LA THESE

La première partie présente le langage APL, en introduisant progressivement les concepts de base les plus intéressants, et le système APL en le définissant comme l'environnement idéal à l'exécution de la notation APL. Puis elle s'intéresse plus particulièrement aux travaux menés depuis 1970 en matière d'interprétation, d'optimisation et de compilation du langage. L'inventaire des problèmes importants liés à sa réalisation sur une machine réelle permet de comprendre les difficultés rencontrées dans la conception d'une véritable machine APL.

La seconde partie introduit une définition très générale des j-vecteurs, dresse la liste des opérations conservant cette structure et étend à cette définition plus large les règles du beating proposées par Abrams. Elle décrit la réalisation des partages de données et plus généralement expose le principe hiérarchique de la mémoire APL et la conception de la gestion récursive des blocs-pointeurs. Elle étudie ensuite un point sensible de tout système APL : l'accès aux objets. Cette étude menée sous trois éclairages différents : celui des opérateurs, celui des variables optimisées et celui de la mémoire virtuelle simulée, permet de dégager de nouvelles optimisations, d'unifier les fonctions d'accès aux objets et conduit à la définition d'un compilateur dynamique d'accès : le générateur de liens.

Enfin, dans la dernière partie, deux extensions sont présentées : les tableaux généralisés et l'approche qui en a été faite, ainsi qu' une panoplie d'outils permettant l'écriture en APL d'applications fermées et sûres. Dans chacun des cas, l'option introduite dans le système est exposée et critiquée.

PARTIE I

APL et l'ETAT de l'ART

Le premier chapitre présente le **langage APL** , en introduisant progressivement les concepts de base les plus intéressants - objets et syntaxe du langage, portée des noms, primitives et opérateurs -, et le **système APL** en le définissant comme l'environnement idéal à l'exécution de la notation APL.

Le second chapitre s'intéresse plus particulièrement aux travaux menés depuis 1970 en matière d'interprétation, d'optimisation et de tentatives de compilation du langage. L'inventaire des problèmes importants liés à sa réalisation sur une machine réelle - représentations internes, gestion de mémoire dynamique, liaison dynamique des noms aux objets lors de l'exécution - permet de comprendre les difficultés rencontrées dans la conception d'une véritable machine APL. Cette étude aboutit à une conclusion très réservée sur l'intérêt de la mise en oeuvre de techniques sophistiquées de compilation dynamique du langage.

C H A P I T R E 1

I - LE LANGAGE APL

Le langage APL est une notation mathématique abstraite née des travaux de K.E. IVERSON et A.D. FALKOFF sur la formalisation des algorithmes (L1,L2,L6,L7). Il est conçu pour la manipulation globale de données structurées en tableaux rectangulaires, denses et homogènes en type. La récursivité, le dynamisme complet des données, dû essentiellement à l'absence de déclaration, et une vaste panoplie de fonctions primitives en font un langage de programmation extrêmement puissant.

Des concepts très simples en rendent l'approche aisée.

I.1 - Les objets du langage

De façon très classique les variables et les programmes sont désignés par des noms symboliques.

I.1.1 - Variables et constantes

Toute donnée est caractérisée par un type et une structure propres. Il n'y a que deux types explicites :

- le type numérique
- le type caractère.

Les constantes sont scalaires. 100 et 1E2 représentent la même constante numérique. 'A' ou ' ' représentent deux scalaires de type caractère. Elles peuvent être aussi vectorielles : 1 ¯5 3 et 'ABC' sont deux constantes de trois éléments.

Une variable est définie à partir du moment où on lui a donné une valeur. Pour cela, on utilise l'opération d'affectation :

<identificateur> ← <expression APL>

Les données ont une structure rectangulaire définie par :

- un rang ou nombre d'indices nécessaires pour isoler un élément individuel de la structure
- une liste de dimensions précisant les domaines de variation de ces indices.

La primitive de restructuration (⍴ dyadique) fabrique des données structurées :

```
      □←LOCO←3 6⍴' □ ∩ ⌈<□□□□□/○○ ○○'

 □ ∩ ⌈
<□□□□□
/○○ ○○
```

```
      □←WAGON←3 9ρ'          +|______|  oo  oo '

+|______|
  oo  oo
```

Le scalaire ou élément unique, constitue le cas particulier d'une donnée de rang nul, n'ayant aucune dimension.

Les variables s'identifient donc aux scalaires, vecteurs, matrices et tableaux de rang supérieur que l'on rencontre en algèbre linéaire.

I.1.2 - Primitives et opérateurs

Grâce à un alphabet très riche, toutes les opérations de base du langage sont représentées par des symboles spéciaux ; ceux-ci nous sont très familiers :

2+3.5 (addition)

÷.2 .5 (inverse)

ou quelques fois, un peu moins :

?7 9 (tirage aléatoire)

3.5⌈A (maximum)

Ces primitives s'appliquent élément par élément à leur(s) opérande(s) ; on parle alors de primitives scalaires :

2 3 4 ⌊1 7 3 réalise le minimum terme à terme entre deux vecteurs de trois éléments. Le résultat rendu sera donc le vecteur 1 3 3.

Ou bien, elles s'appliquent de façon globale comme dans cet exemple tiré de (L4) illustrant l'utilisation de la primitive de concaténation :

```
      □←TRAIN←LOCO,WAGON,WAGON

 □ ∩ ⌈
<□□□□□+|______|+|______|
/oo oo  oo  oo   oo  oo
```

Le principe d'extension des scalaires rend leur emploi agréable. Ainsi :

1.20×*PRIX*←100 150 180 200

effectue un calcul d'intérêts annuels sur un ensemble de prix.

Des opérateurs permettent de modifier l'action des primitives scalaires. La réduction, par exemple :

```
      +/PRIX
630
```

D'autres opérateurs permettent de combiner entre elles deux primitives scalaires. Voici un calcul d'intérêts croisés :

```
      1.2  1.3∘.×100  150
120  180
130  195
```

P ← A+.×B est un produit interne d'addition-multiplication (si A et B sont des matrices, P représente alors leur produit matriciel classique). Certaines primitives ne s'intéressent qu'à la structure des opérandes. Ainsi, ρ appliqué à un vecteur donne le nombre d'éléments de ce vecteur :

```
      ρPRIX
 4
```

En fait, d'un point de vue conceptuel, un opérateur s'applique sur des primitives en rendant comme résultat une nouvelle primitive. Par exemple, le langage étant pourvu de 21 primitives scalaires, on obtient au total 441 produits internes différents.

Primitives et opérateurs permettent d'écrire de façon très concise des expressions reflétant clairement le résultat cherché. Ainsi, le calcul de la moyenne du vecteur PRIX s'écrit :

```
      (+/PRIX)÷ρ PRIX
157.5
```

I.1.3 - Fonctions définies

Si, pour un certain problème, l'utilisateur a besoin d'une primitive particulière, et qu'il ne la trouve pas dans le langage lui-même, il lui est possible de la définir sous la forme d'une fonction "utilisateur" ou "définie". Pour obtenir l'opération "moyenne-de" présentée ci-dessus, il écrira :

```
      ∇R←MOY V
[1]   R←(+/V)÷ρV
[2]   ∇
```

Ensuite, il l'utilisera exactement comme une primitive standard du langage :

```
      2.5+ MOY PRIX
  160
```

I.2 - Syntaxe du langage

Nous ne donnerons pas une description complète et détaillée de la syntaxe formelle du langage APL. Il existe maintenant des documents très précis (L11) tentant de définir un standard. Nous rappellerons simplement les principes syntaxiques de base qui permettent une utilisation immédiate du langage.

Toutes les instructions APL s'évaluent de la droite vers la gauche **sans priorité** entre les opérateurs. En notation B.N.F. (Backus Naur Form), on écrira la syntaxe d'une expression :

```
<expression> ::= <opérande>
                 | <fonction-monadique> <expression>
                 | <opérande> <fonction-dyadique> <expression>
                 | <fonction-niladique>
```

Un opérande pouvant être à son tour :

```
<opérande>  ::= <constante>
                | <variable>
                |(<expression>)
```

Cette définition, volontairement simplifiée, ne tient compte ni des expressions indicées ni de la syntaxe particulière des opérateurs. Elle introduit cependant la notion de **valence** , une primitive, un opérateur ou une fonction définie pouvant avoir un, deux ou aucun argument(s).

Ainsi, seule la syntaxe utilisée permettra de distinguer l'exponentielle :

```
      *1
2.781828
```

de l'élévation à une puissance :

```
      2*3
8
```

Dans la suite de l'exposé, pour supprimer toute ambiguïté, nous préciserons la valence de la primitive en indice :

$*_1$ représentera l'exponentielle et $*_2$ l'exponentiation.

I.3 - Portée des noms - Globalité - Localité

Sans mention spéciale contradictoire, les noms symboliques ou identificateurs des objets ont tous la même portée : ils désignent des fonctions ou des variables globales.

Un objet peut cependant être défini comme local à une fonction. Cette particularité prend son importance au moment de l'exécution de la fonction : à son appel, tous les noms locaux qui y ont été définis "masquent" ou "ombragent" les objets de mêmes noms des niveaux appelants. Au retour de la fonction, ces objets redeviennent accessibles. Un programme APL - entendons, une fonction - a une **structure de bloc statique** , liée à la déclaration des identificateurs locaux : paramètres, résultat, étiquettes et variables locales explicitement nommées.

La règle de portée des noms et le mécanisme de protection des objets à l'exécution introduisent la notion de **structure de bloc dynamique** , liée elle, à l'exécution et permettent en conséquence la création de fonctions récursives.

A titre d'exemple, nous redéfinirons récursivement la fonction factorielle, $!_1$ (L5) :

```
    ∇  R←FACT N
[1]    →0×⍳N=R←1
[2]    R←N×FACT N-1
    ∇
```

Les noms implicitement locaux N et R désignent des objets différents dans chaque bloc dynamiquement créé. Nous supposerons la fonction activée, puis interrompue avant son premier retour (fig.1).

```
      FACT 3
6
```

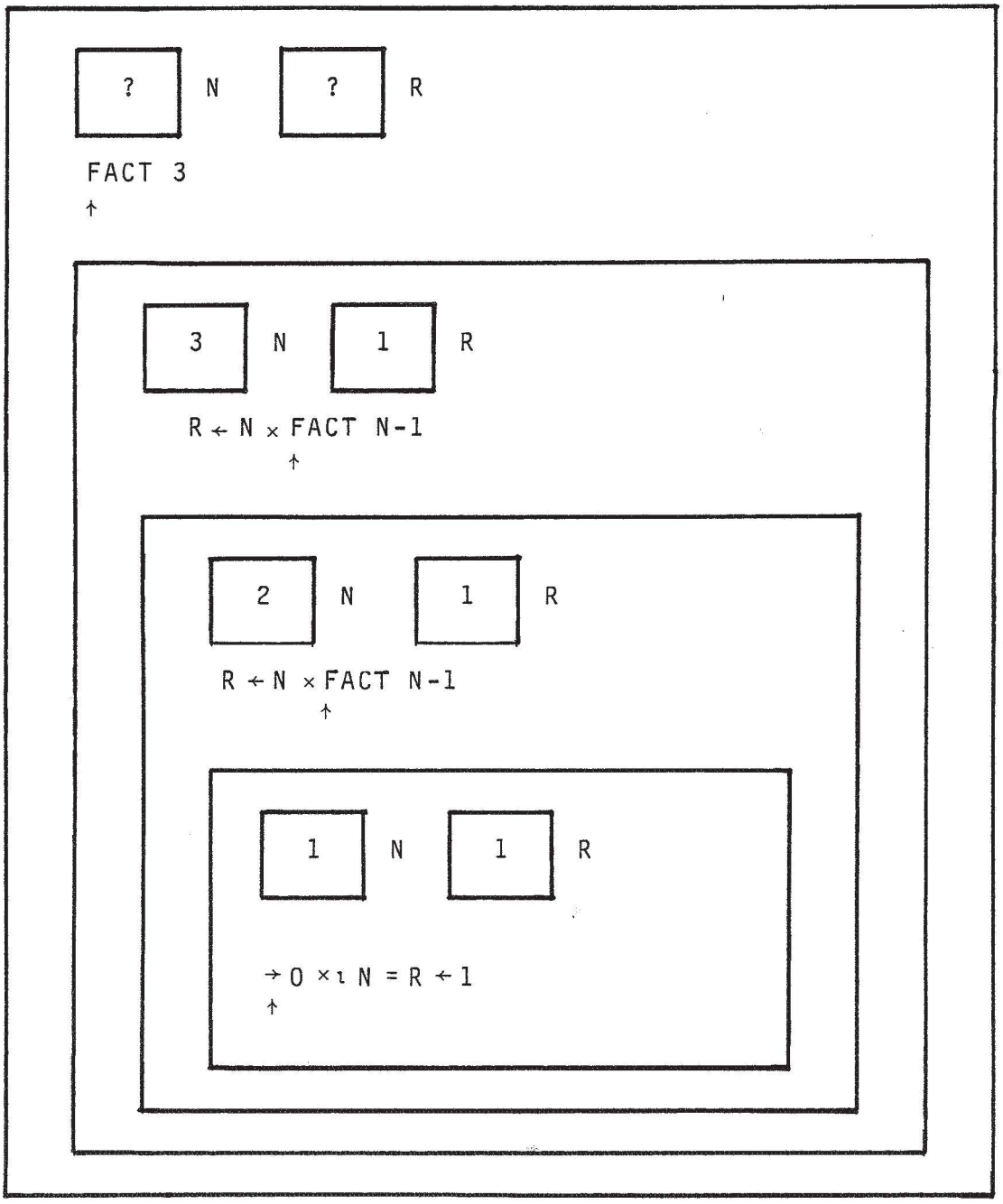


Figure 1 : Blocs dynamiques

I.4 - Classification des primitives et opérateurs

Le langage est caractérisé par une collection importante de primitives. Des propriétés communes permettent leur regroupement en classes.

I.4.1 - PRIMITIVES SCALAIRES : elles modifient les valeurs sans altérer les structures.

Arithmétiques : $+_1, \div_1, -_1, \times_1, *_1, \circledast_1, |_1, !_1, \circ_1, \lceil_1, \lfloor_1$
$+_2, \div_2, -_2, \times_2, *_2, \circledast_2, |_2, !_2, \circ_2, \lceil_2, \lfloor_2$

Comparatives : $<_2, \leq_2, \neq_2, =_2, \geq_2, >_2$

Logiques : $\sim_1, \wedge_2, \vee_2, \overset{\sim}{\wedge}_2, \overset{\sim}{\vee}_2$

I.4.2 - PRIMITIVES DE RESTRUCTURATION : elles modifient les structures sans altérer les valeurs.

Linéarisation : $,_1$ - Concaténation : $,_2$ $\overset{-}{,}_2$ - Laminage : $,_2[d]$

Renversement : $\ominus_1, \phi_1$ - Rotation : $\ominus_2, \phi_2$

Compression : $/_2, \not/_2$

Expansion : $\backslash_2, \not\backslash_2$

Transposition simple : $\circledcirc_1$ et généralisée : $\circledcirc_2$

Prend : $\uparrow_2$

Laisse : $\downarrow_2$

Restructuration : ρ_2

Indexation : $[;;;]_1$

I.4.3 - PRIMITIVES MIXTES ET SPECIALES : elles correspondent à toutes les autres primitives.

Tris : $\grave{\Delta}_1$ et $\grave{\nabla}_1$

Tirages aléatoires : $?_1$ et $?_2$

Numération : $\perp_2$ et $\top_2$

Affectation simple : $\leftarrow_2$ et indicée : $[;;;]\leftarrow_2$

Contrôle d'exécution : $\rightarrow_0, \rightarrow_1$ et $\hat{\pm}_1$

Génération d'indices : ι_1

Appartenances : $\in_2$ et ι_2

Dimension : ρ_1

Formattage : $\overset{\top}{\ }_1$ et $\overset{\top}{\ }_2$

Inversion de matrice : $\boxplus_1$ et résolution de système : $\boxplus_2$

Certaines de ces primitives intègrent directement au langage des outils que l'on rencontre d'ordinaire dans des bibliothèques de programmes : tris, inversions de matrice, générateur de nombres aléatoires.

L'exécution d'une chaîne de caractères ($\mathbin{⍎}_1$) en tant que phrase du langage fait d'APL un macroprocesseur.

I.4.3 - OPERATEURS : ils modifient ou combinent l'action des primitives scalaires (f_2 et g_2).

Réduction : $f/_1$

Balayage : $f\backslash_1$

Produit externe : $\circ.{}_2f$

produit interne : $f.{}_2g$

II - LE SYSTEME APL

Beaucoup plus qu'un langage de haut niveau, APL apparaît comme un outil logiciel d'une rare puissance car il s'insère dans un système complet et cohérent.

Un processeur FORTRAN ou COBOL repose sur le système hôte pour de nombreux services. A l'opposé, les implantations d'APL fournissent non seulement le support d'exécution de la notation APL, mais encore un éditeur de fonctions, un système de mise au point, une gestion de bibliothèques de programmes. l'accès à des fichiers extérieurs. L'ensemble constitue un système autonome que l'utilisateur conçoit comme une machine APL (fig.2).

Les premières réalisations d'interprète sur IBM/1130, puis sur IBM/360 furent l'occasion pour leurs auteurs de concevoir une véritable machine APL (L6,L7,L8). Les buts poursuivis étaient nombreux :

- rendre cette machine la plus accessible possible en éliminant tout ce qui décourage le non informaticien de la programmation (langage de contrôle ésotérique, nombreuses étapes intermédiaires, difficultés de communication des programmes avec le monde extérieur)
- obtenir une indépendance du calculateur hôte, sur lequel est réalisée cette machine, la plus grande possible. Ce souci comprend la recherche de la transparence à l'utilisateur des contraintes matérielles de la réalisation (taille de la mémoire centrale, longueur des mots utilisés, représentations numériques, code des caractères ...)
- intégrer harmonieusement au langage tous les services satellites de la programmation que l'utilisateur est en droit d'attendre : gestion des programmes (édition, mise au point, archivage), contrôle complet de l'environnement de travail et d'exécution des programmes
- confier le plus de travail automatique et de décisions très ponctuelles à la charge de cette machine et non pas à celle de l'utilisateur.

Pour ce dernier, cette recherche du confort réduit à l'essentiel l'investissement intellectuel parasite que nécessite en général la programmation d'un problème très simple.

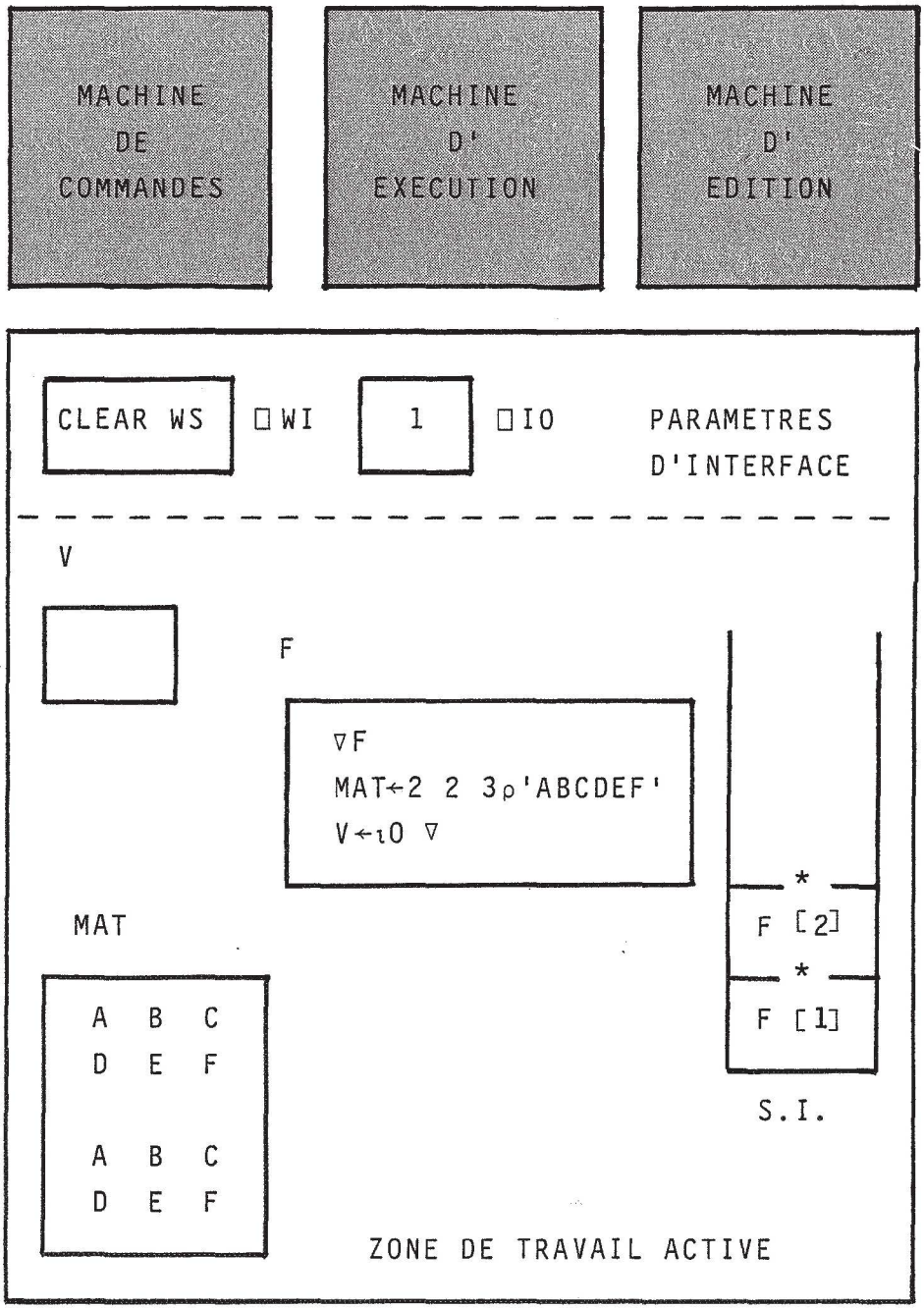


Figure 2 : La machine APL

Ces efforts ont conduit les auteurs à la définition d'un système qui, depuis, sert de standard (L6). Ses principales caractéristiques qui répondent aux objectifs sont :

II.1 - ASPECT INTERACTIF

D'une façon analogue à BASIC ou LSE, l'interprète APL travaille dans deux modes :

a) Mode "exécution" : toute ligne présentée à l'interprète est immédiatement décodée, analysée puis évaluée. Le résultat, s'il existe, est imprimé sur le terminal. Le système choisit dans ce cas le format de représentation externe des nombres et des structures le plus simple et le plus clair possible.

```
      1+2 2⍴⍳4
 2  3
 4  5
```

b) Mode "définition de fonction" : un symbole spécial (∇) permet d'entrer et de sortir de ce mode. Toute ligne introduite dans ce mode est uniquement mémorisée pour exécution ultérieure. On peut aussi détruire ou modifier des lignes existantes, imprimer des fonctions (G9).

II.2 - LA ZONE DE TRAVAIL ACTIVE

Le contexte de travail s'appelle zone de travail active. Vue de l'utilisateur, cette zone représente un "cahier de brouillon" dans lequel il conserve ses variables, construit et met au point ses fonctions. Quand l'ensemble lui paraît satisfaisant, il peut sauvegarder cet espace actif sous une forme que l'on appelle zone de travail inactive.

Une zone de travail contient au yeux de l'utilisateur :

- la table des symboles contenant tous les noms qu'il crée
- les variables et les fonctions qu'il définit
- le vecteur d'états (encore appelé pile des contextes) qui est constitué par l'empilement des blocs dynamiques d'exécution.

De plus, cette zone de travail est sous le contrôle d'un

certain nombre de paramètres servant d'interface entre le système APL et la zone active. Ces paramètres correspondent à des objets communs au système et à l'utilisateur. Ce sont des variables système que tous deux peuvent explicitement consulter ou modifier :

- □IO, l'origine des indices
- □CT, la tolérance des comparaisons, etc.,

ou des paramètres implicites, dans le sens où leur modification ne peut se faire directement par une affectation mais par l'emploi d'une commande système :

-)DEPTH 60 limite la profondeur du vecteur d'états à 60 appels de fonction
-)SYMBOLS 200 détermine le nombre maximum de symboles utilisables.

II.3 - SURVEILLANCE DE L'EXECUTION

La mise au point des programmes est facilitée par l'aspect interactif mais aussi par l'aspect interprétatif de l'exécution d'une ligne APL. La définition de la machine APL est prolongée par l'inventaire complet des erreurs de cette machine, l'utilisateur n'ayant jamais à connaître les erreurs pouvant survenir aux différents niveaux de réalisation inférieurs (G10).

Selon leur visibilité, variables et fonctions restent accessibles par leur nom externe.

Le vecteur d'états est accessible en permanence.

Il est possible de définir des points d'arrêt ou de trace dans les fonctions définies.

L'exécution et les échanges sont interruptibles. Les interruptions de la machine APL ne sont prises en compte évidemment qu'aux moments où l'état de cette machine est observable, c'est à dire entre deux instructions APL. La reprise de l'exécution est donc possible.

Le système de recouvrement des erreurs très sophistiqué mis en oeuvre dans APL permet presque toujours au programmeur de retrouver tous les objets de la zone active dans l'état précédant l'erreur.

Les rapports d'erreur sont clairs et circonstanciés. La machine APL procède en outre à de sévères contrôles dynamiques à l'exécution :

```
INDEX ERROR
F[3]  A←C÷B[I;]
                ^
      I
37
      ρB
20  30
```

II.4 - COMMUNICATIONS AVEC LE MONDE EXTERIEUR

II.4.1 - ECHANGES EXPLICITES : Deux variables système □ et ⍞ représentent des fenêtres sur le monde extérieur. La syntaxe d'un échange ne peut être plus simple :

□←*A* symbolise la sortie de la variable A vers le terminal

B←□ demande l'acquisition depuis l'extérieur des valeurs qui constitueront B.

Le système déterminera lui-même le format idoine. Un opérateur spécialisé (⍕) autorise cependant des formattages plus précis.

II.4.2 - ECHANGES IMPLICITES : Par convention, lorsque le résultat d'une expression n'est pas affecté, le système l'imprime sur le terminal, réagissant ainsi comme si l'affectation était faite à la variable □. Connaître le contenu d'une variable se formule comme une question élémentaire (A?) :

```
      A
2 3 5
```

II.5 - ARCHIVAGE DES ZONES DE TRAVAIL

La sauvegarde de la zone de travail active consiste à prendre une image de celle-ci - de tous les objets qu'elle contient (table des symboles, variables, fonctions, pile des contextes, paramètres système) - et à la conserver sous un nom donné sur un support non volatile : on crée une zone de travail inactive.

Une zone de travail inactive peut être chargée à la place - il y a écrasement - de la zone active. On récupère alors le travail complet dans l'état où il était lors de sa sauvegarde. On peut ainsi reprendre l'exécution où elle était arrêtée.

Des transferts plus fins permettent d'amener en zone active des objets d'une zone inactive (commandes de copie d'objets).

II.6 - FICHIERS

Nous terminerons en ajoutant qu'un système APL n'est réellement complet que s'il offre une gestion de fichiers, que ces fichiers soient accessibles directement - système de fichiers spécialisés APL (F1) - ou indirectement - emploi d'une variable partagée (L8) -.

CHAPITRE 2

I - PROBLEMES GENERAUX INHERENTS A LA DEFINITION DU LANGAGE

I.1 - Les caractéristiques déterminantes du langage

Elles résultent des trois options de base du langage APL :

- dynamisme complet des données. Ce dynamisme est dû à l'absence de déclaration de taille ou de type des variables, les attributs d'une variable pouvant être modifiés à chaque réaffectation de cette variable au moment de l'exécution.

- puissance des opérations du langage. Celles-ci s'appliquent à des tableaux complets et non pas à de simples éléments.

- absence de hiérarchie statique dans la définition des fonctions. Toute fonction définie est un "programme principal" en puissance. La localisation et la création dynamique de fonction (par □FX, ou par copie) entraînent qu'une même phrase APL, prise dans des contextes d'exécution distincts, peut contenir un nom représentant une fonction ayant dans chacun des cas une valence différente.

Sur une machine réelle, on voit qu'il faudra prévoir :

- des routines de gestion dynamique de la mémoire de travail

- un système d'itération pour le traitement des opérations APL au moyen des instructions disponibles, sachant que l'existence sur la machine d'un processeur de calcul vectoriel ne nous en dispensera pas

- un mécanisme d'adressage symbolique, l'espace des objets APL étant un espace de noms plutôt qu'une mémoire vectorielle au sens classique du terme. Nous appelerons mémoire de la machine APL ou en abrégé mémoire APL un tel espace symbolique.

I.2 - La machine APL

I.2.1 - Définition :

Une machine APL est un ordinateur dont le matériel est susceptible d'exécuter automatiquement un code machine très proche du langage APL. Elle est donc capable d'exécuter des opérations comme :

- la gestion de la mémoire APL en fonction des besoins de l'unité centrale

- l'adressage symbolique des objets APL, l'unité centrale

étant consciente non seulement de leur **nom** , mais également de leur type, de leur rang et de leurs dimensions. Nous appelerons **attributs** de l'objet APL l'ensemble de ces informations.

Le code machine reste très proche du langage APL, la machine étant, par exemple, une machine à pile.

I.2.2 - Exemple de fonctionnement :

1°) Soit à exécuter l'instruction APL : R←A + B

2°) La traduction dans le "langage machine" se résume à une modeste analyse lexicographique :

début id"B" + id"A" ← id"R" fin

Le code machine est généré à l'envers pour préparer l'exécution selon la règle APL. Les unités lexicales reconnues sont décrites par un doublet (type, valeur) :

- début, id, +, ←, fin sont des codes instruction de la machine,

- "A", "B", "R" sont des opérandes symboliques (sur une machine classique, ce seraient des adresses).

3°) L'exécution du code se décompose ainsi :

Phase	Unité	Pile	Action
0	début	findepile	initialiser la pile.
1	id"B"	findepile	B est reconnu comme variable. La référence à B est empilée.
2	+	ref"B" findepile	Empiler l'opération + .
3	id"A"	+ ref"B" findepile	A est reconnu comme une variable. L'opération + est dyadique et exécutée. Le résultat ξ est mis en haut de pile

Phase	Unité	Pile	Action
4	←	ref"ξ" findepile	Empiler l'opération ← .
5	id"R"	← ref"ξ" findepile	L'opération ← est exécutée, elle transforme la référence "ξ" en la référence "R".
6	fin	ref"R" findepile	Le sommet de pile est dépilé. L'opération exécutée est ←, il n'y a pas d'impression
7	fin	findepile	Arrêt ou passage à l'instruction suivante.

I.3 - La machine naïve

Elle est réalisée au moyen d'un interpréteur exécutant naïvement et séquentiellement des actions comme celles décrites précédemment ; il ne tient pas compte des optimisations possibles.

Dans un premier temps, on peut la considérer comme une simulation sur une machine existante d'une machine APL. Sur la machine existante, il faudra alors prévoir et écrire des modules réalisant les fonctions suivantes :

- pour la gestion de mémoire, allocation et libération dynamiques de zones de mémoire ; il se pose alors le problème de la représentation de la mémoire APL

- pour la gestion des noms et la codification des lignes APL : passage des noms externes en noms internes et génération de code machine

- pour l'exécution des instructions : décodage du code machine, accès aux opérandes, choix des décisions d'action, un module de traitement par opération APL.

La plupart des interprètes APL sont naïfs. On en trouve de

nombreuses descriptions dans la littérature (R1,R2,R3).

II - PROBLEMES DE REPRESENTATION INTERNE

La machine APL a trois types distincts d'objets à gérer :

a) les données

b) les fragments de code machine

c) ses propres tables internes.

Dans un premier temps, nous laisserons de côté le problème des tables internes dont nous ignorerons a priori le rôle, la structure et même le nombre.

II.1 - Représentation des données

Aucune contrainte n'existe dans la définition du langage sur la représentation interne des données manipulées par la machine. Dans la réalité, deux types d'informations sont nécessaires pour décrire une donnée APL.

La nature de la donnée définie par l'utilisateur : c'est le type explicite. Sur APL/360, on distingue les données numériques et les données caractères (L7,L8).

La façon dont ces données sont conservées dans la machine : c'est l'information donnant une signification au champ de bits représentant la donnée.

Si la première doit toujours rester connue de l'utilisateur, la dernière peut être choisie par la machine pour optimiser certaines de ses ressources.

En général, le type caractère, que l'on rencontre dans les réalisations existantes, est représenté sur un octet en utilisant un code spécifique : ASCII ou EBCDIC ou encore un code interne mieux adapté aux opérations que l'on désire effectuer sur la machine : "Z-code", comme sous APL/360 ou APL/16 (R3,R2,L9).

Le type numérique est en général représenté de plusieurs manières. Par exemple, on codera :

- un logique ou booléen sur un bit
- un entier sur un mot de la machine
- un nombre réel sur une représentation flottante de la machine.

II.2 - Intérêt et coût des représentations multiples

II.2.1 - Choix des représentations

Il n'y a pas d'argument déterminant dans ce choix. Il faut cependant considérer les points suivants :

- l'utilisateur apprécie d'avoir 16 chiffres significatifs, voire plus (U2) ; les gestionnaires comptent "au centime près" et réclament une grande précision ; les scientifiques préfèrent pouvoir manipuler de grands nombres (1E100) ou des petits (1E~100), ils calculent eux-mêmes leurs taux d'erreur
- la multiplicité des représentations internes permet :
  . un gain de place dans les zones de travail et les fichiers
  . un gain de temps à chaque fois qu'il y a adéquation des représentations aux calculs demandés
- cette même multiplicité entraîne en revanche :
  . une perte de place et une complexité accrue dans l'écriture de l'interprète pour les accès, les conversions
  . une perte de temps dans certains cas (conversions implicites par exemple)
- l'unicité des représentations numériques conduira aux conclusions duales.

II.2.2 - Les problèmes de conversion

Conversions implicites : un calcul comme A+B, où A est un vecteur entier et B un vecteur réel, va nécessiter des opérations de conversion pour effectuer le calcul élément par élément. A, ici, sera converti en réel, ce qui est toujours possible. En revanche, un calcul comme ~X, où X est réel, va demander de vérifier si tous les éléments de X valent bien 0 ou 1. On admet en outre une certaine imprécision, afin que des valeurs voisines de 0 ou 1 soient admises, alors que les autres provoqueront une erreur.

Conversions explicites : elles sont entraînées par certaines primitives (partie entière en est une) et posent souvent des problèmes de réalisation. Ainsi, si le nombre X vaut 375243.7225, sa partie entière par défaut est mathématiquement égale à 375243. Pourtant, avec une arithmétique entière 16 bits, elle ne sera pas calculable par une simple conversion de flottant à entier...

Quelles que soient les solutions choisies, les problèmes de conversion restent délicats à résoudre, et sont souvent la cause

d'imperfections des systèmes APL (G1,U3).

II.3 - Code machine APL

II.3.1 - Choix du langage intermédiaire

Le problème du choix d'un code machine, évoqué dans I.2, est le problème classique du langage intermédiaire.

Le principe est le suivant : étant donné une ligne source APL, **indépendante de tout contexte** , on recherche les transformations susceptibles d'en accélérer l'exécution à chaque fois que celle-ci sera demandée par l'utilisateur, les lignes des fonctions définies étant les plus intéressantes pour de telles transformations. Manifestement, plus ces transformations sont profondes, plus le code intermédiaire obtenu devient dépendant du contexte et plus il s'éloigne du code idéal de la machine APL.

Prenons, par exemple la ligne APL suivante :

$$PN\leftarrow PN-B\leftarrow\sim(QX=1)\wedge(A=B)\vee('A'\in OS)$$

1ere transformation : Découpage de la ligne et reconnaissance des unités lexicographiques :

Identificateurs : PN, B, QX, A, OS

Opérateurs et séparateurs : ←, -, ~, =, ∧, ∨, ∈,), (

Constantes numériques et caractères : 1, 'A'

Cette première transformation est indépendante du contexte et permet de reconnaître les unités mal formées.

2eme transformation : Codage des constantes dans les représentations internes choisies. La machine hébergeant l'interprète d'une part, et les choix de représentation de ses auteurs d'autre part commencent à intervenir.

3eme transformation : Codification des noms symboliques externes en noms symboliques internes. En général, les noms internes utilisés sont des numéros de symbole. Ce numéro permettra d'accéder très rapidement au descripteur ou ensemble des attributs de l'objet que le symbole désigne. C'est donc par la même occasion un numéro de descripteur.

Une telle transformation évitera des recherches dans la table des symboles, recherches coûteuses en temps si cette table est grande - et en particulier, si elle est segmentée ou située dans un

espace virtuel -. Le code devient alors très dépendant du contexte, en ce sens que pour utiliser la fonction dans un espace de travail différent, il faudra mettre à jour dans le code tous les numéros des symboles. Ce cas se présente quand on copie dans la zone active des fonctions définies d'une zone inactive, les attributions de numéros internes aux mêmes symboles externes n'ayant aucune raison d'être identiques. La probabilité d'une telle opération reste très faible cependant devant celle de l'emploi répété d'une fonction en zone active.

4eme transformation : Codification des idiomes de la ligne et attributions de valence aux primitives, aux opérateurs et aux fonctions.

On appelle **idiome** une forme sentencielle caractéristique du langage très fréquemment utilisée. La reconnaissance et la codification de certains idiomes sous la forme de primitives "internes" plus performantes permettent d'économiser encore un peu du temps d'exécution. On trouvera chez Perlis une étude statistique détaillée des idiomes APL les plus classiques (I7).

Exemples de codification d'idiomes :

- la suite ⍴ ⍴ codée en rang de
- l'instruction id"I" ← id"I" + const"1" devient incr id"I"

Certaines réalisations vont beaucoup plus loin en essayant de reconnaître dès l'analyse des constructions comme la réduction (⍟/) ou le balayage (⍟\), les produits internes et externes, l'indexation des variables et des opérateurs (R4). Des attributions de valence aux primitives sont tentées, c'est à dire que l'on essaie de préciser si les primitives sont monadiques ou dyadiques.

Il faut cependant prévoir une complément d'analyse syntaxique au moment de l'exécution, comme le montre l'exemple très simple suivant :

```
    ∇ R←F
[1]   R←A+B
    ∇
```

Les informations dont on dispose, concernant les objets A et B, à la codification de la ligne ne permettent **absolument pas** de savoir ce que désigneront ces noms à l'exécution. Ce problème est

connu sous le nom de **liaison dynamique des noms symboliques aux objets** (G2).

En l'occurence, la fonction F peut tout aussi bien être appelée dans un contexte où A est une fonction monadique, ou dans un contexte où elle est dyadique, voire niladique. Dans chacun des cas, nous obtenons :

- l'appel de + monadique
- une erreur de syntaxe, A étant dyadique
- l'appel de + dyadique, A niladique devant fournir un résultat.

II.3.2 - L'état de l'art

Les travaux actuels concernant ce point précis ont pour but de trouver des représentations internes de plus en plus proches d'une forme compilée.

On citera les travaux de Battarel, Delbreil et Kalfon par exemple (I1,I2,I3,I4), qui utilisent les principes suivants :

Une ligne APL est conservée sous sa forme externe.

A la première exécution de cette ligne, du code machine est généré, code étroitement lié à la nature, au rang des variables, à la syntaxe déposée, au moment de cette génération, des fonctions.

A l'issue de cette première exécution, le code reste lié à la ligne APL correspondante, en même temps que des indicateurs qui permettent de le valider.

Au cours des exécutions ultérieures de la ligne, le code machine est réutilisé si les variables n'ont pas changé de nature ou de rang, ou si les fonctions ont la même syntaxe...

Ces méthodes n'en restent malgré tout qu'au stade expérimental, et il n'est pas encore prouvé qu'elles apportent un grand gain par rapport aux méthodes classiques.

III - LA GESTION DE MEMOIRE

III.1 - Gestion de la mémoire uniforme

C'est le cas le plus simple de la gestion de mémoire à un niveau : toute la zone active de l'utilisateur est présente en mémoire principale (comme sous APL/360 ou sous APL-SV). Sauf un certain nombre de tables situées à adresses fixes, la mémoire est organisée

sous forme de listes simplement ou doublement chaînées. En général, deux listes indépendantes recouvrent toute la mémoire principale utilisable : la liste des blocs libres et celle des blocs occupés. Les algorithmes restent très classiques : First Fit, Best Fit, etc. (G3, G4). Les blocs contenant des données peuvent être déplacés à des fins de recompactage de la mémoire centrale (techniques de ramasse-miettes).

Deux primitives seulement caractérisent ce genre de gestion et suffisent à l'interprète :

- A := Obtenir-mémoire ;
- libérer-mémoire (A) ;

A étant une adresse de la mémoire.

III.2 - Gestion de mémoire hiérachisée

La hiérarchie est composée en général de deux niveaux de mémoire et est imposée par le manque d'espace en mémoire principale. On utilise alors une segmentation, données, tables et programmes de l'interprète pouvant se trouver indifféremment en mémoire principale ou en mémoire secondaire. Les disques à têtes fixes ou à têtes mobiles servent le plus souvent de mémoire secondaire. Là encore, les algorithmes sont très classiques (G3,G4,M1,M2). Des solutions plus adaptées à APL sont décrites dans (M3,R2).

III.3 - Gestion de mémoire virtuelle

La différence essentielle existant entre les principes mis en jeu et ceux exposés en III.2 est la même que celle qui existe entre la pagination et la segmentation. En effet, de telles réalisations d'APL vont tirer partie de dispositifs matériels existants sur la machine hôte, et permettent à un programme d'utiliser tout le champ d'adressage prévu dans les instructions machine (typiquement 2^{24} octets dans les gros systèmes, soit 1 677 216 octets). La gestion de la mémoire, en termes de primitives, est alors la même que dans III.1.

On ne connait guère sur un petit ordinateur qu'une unique implantation d'APL de ce genre, réalisée sur un MITRA-15 (I5). Disons pour simplifier que ce système simule de manière logicielle un dispositif de mémoire virtuelle segmentée, la différence avec III.2

étant que cette fois une donnée APL peut être partagée en plusieurs segments. La gestion de mémoire peut gérer 2^{15} segments, chaque segment pouvant contenir 2^{13} octets. Les objets gérés par cette dernière sont organisés sous forme d'un graphe orienté, sans circuit.

IV - LES METHODES D'OPTIMISATION

De nombreuses recherches visant à optimiser l'exécution d'APL ont été menées depuis 1970 pour les raisons suivantes :

- plus que dans un autre langage, l'utilisateur a tendance à rédiger des expressions très concises (afin, par exemple, d'éviter la programmation de boucles), ce qui entraîne souvent des opérations inutiles, ou coûteuses en temps de traitement et en place mémoire
- pour un même problème, les coûts en unité centrale peuvent être dix à cent fois plus grands en APL que dans d'autres langages.

Nous classerons en trois grands groupes, par ordre de difficultés de mise en oeuvre croissantes, les différentes méthodes d'optimisation :

- algorithmes ponctuels spécialisés
- méthodes introduites par Abrams (I6) et méthodes dérivées
- essais de compilation.

IV.1 - Algorithmes spécialisés

Une optimisation locale est le traitement d'un cas particulier du domaine d'un opérateur, faisant que, dans ce cas particulier, l'opérateur nécessite moins d'espace mémoire ou de temps d'unité centrale pour son traitement.

Exemple 1 : A*B, $A*B \leftrightarrow e^{B\text{Log}A}$

1°) cas général : $A*B \leftrightarrow *Bx\circledast A$

2°) cas d'un exposant entier positif : $B \in N^*$

$A*B \leftrightarrow AxAx...xA$ (B fois)

L'opération peut se faire en (B-1) multiplications, ou mieux en ($\log_2 B$) multiplications, d'où un gain de temps.

Exemple 2 : xA donne les signes des éléments de A :

1 si A[I]>0, 0 si A[I]=0, ¯1 si A[I]<0

Le résultat est un vecteur d'entiers de N éléments, que l'on représentera sur N mots. Mais, si A[I]≥0, pour tout I, alors le

résultat peut être codé sur N bits, d'où un gain d'espace.

Exemple 3 : f/A ↔ A[1]fA[2]f...fA[N], est une réduction de A par l'opérateur f. Elle nécessite (N-1) opérations élémentaires f pour son calcul. Le balayage se définit à l'aide de la réduction :

f\A ↔ V, (ρV)↔(ρA) et pour tout I≤N

V[I] ↔ f/I↑A ↔ f/A[⍳I].

L'évaluation de V demande dans le cas général (Nx(N-1)/2) opérations élémentaires f.

Cependant, si l'opérateur f est associatif, il est alors possible de calculer tous les éléments de V en (N-1) opérations en appliquant la formule itérative : V[I+1] ↔ V[I]fA[I+1]. Le gain de temps est important et possible avec les opérateurs fréquemment utilisés dans des balayages : + , x , ∧ , ∨ .

Exemple 4 : Une expression comme ∧/A (resp. ∨/A) vaut 0 (resp. 1) s'il y a dans A au moins un élément de valeur 0 (resp. 1). Une optimisation consiste à provoquer une sortie irrégulière d'itération et à arrêter les calculs dès que l'on rencontre un tel élément.

On trouve, par ailleurs, dans la littérature spécialisée de nombreuses méthodes d'optimisation d'opérateurs "gourmands" comme ~ , ∈, ⊥, ⊤, $?_1$, $?_2$, ⌸ (I8,I9,I11). La littérature générale donne également des algorithmes pour les opérations de tris ⍋, ⍒ ou d'inversions de matrice et de résolution de systèmes linéaires (G3).

IV.2 - Méthodes d'Abrams

Ces méthodes ont été décrites pour la première fois dans (I6), sous les noms de "beating" et de "drag along". Elles ont ensuite donné naissance à des algorithmes spécialisés comme dans (I12) ou plus généraux comme dans (I2) dont nous dirons quelques mots par la suite.

IV.2.1 - Beating et Drag Along

Ce sont les deux notions fondamentales introduites par Abrams. De nombreux essais d'optimisation ont été tentés à partir de ces méthodes. Exposons les brièvement.

1°) - **Le beating** :

Le beating est l'opération consistant à appliquer certaines primitives de sélection non pas sur les valeurs des objets APL, mais sur les descripteurs de ces objets. Précisons quelques termes afin de pouvoir donner des exemples de beating.

- Descripteurs, fonction d'accès : Une collection d'objets élémentaires peut être représentée par un ensemble de positions de mémoire. La méthode la plus courante consiste à utiliser des positions de mémoire **contiguës** . Ainsi, le vecteur V dont les éléments ont pour valeur 12 135 6 0 4 peut être représenté par :

12	mémoire	2004
135	"	2005
6	"	2006
0	"	2007
4	"	2008

La première mémoire utilisée sert en général de repère pour le vecteur V tout entier. L'adresse des autres éléments se calcule en prenant l'adresse de base à laquelle on ajoutera le numéro d'ordre (ou quelque chose de proportionnel) de l'élément désiré :

adresse V[3] ↔ 2004+2 =2006

Dans le cas d'un objet multidimensionnel, une méthode consiste à représenter cet objet **linéarisé** , avec par exemple la convention que le premier indice varie le moins vite quand on suit cette forme linéarisée. Ainsi, la matrice M, de dimensions 2 3, de valeurs :

6	5	12
19	0	2

sera représentée en mémoire par la succession :

6	mémoire	1040
5	"	1041
12	"	1042
19	"	1043
0	"	1044
2	"	1045

Lorsque l'interprète doit manipuler de tels objets, dont les caractéristiques lui sont inconnues jusqu'au moment de l'exécution, il doit être capable de trouver celles-ci dans un descripteur. Le descripteur de V comportera les renseignements suivants :

rang : 1
dimension : 5
adresse : 2004

Celui de M sera :

rang : 2
dimensions : 2 3
adresse : 1040

Pour accélérer l'exécution, on peut conserver également le **vecteur de poids** (G11) : c'est un vecteur W dont le nombre d'éléments est égal au rang de l'objet A décrit, et dont l'élément W[I] est le **pas** permettant de passer de l'élément A[...;k;...] à l'élément A[...;k+1;...]. (ie indice)

Exemple : Matrice M
dimensions : 2 3
vecteur de poids : 3 1
distance de M[0;0] à M[0;1] : 1
de M[0;0] à M[1;0] : 3

Accès à M[I;J] : le calcul 3xI+1xJ donne le déplacement à ajouter à l'adresse de base du tableau pour obtenir l'adresse de l'élément désigné. Un tel algorithme s'appelle **fonction d'accès** .
Ainsi l'adresse de M[1;2] est 1040+1x3+2x1=1045, l'élément trouvé est 2.

Voici donc le nouveau descripteur associé à M :
rang : 2
dimensions : 2 3
vecteur de poids : 3 1
adresse : 1040

Nous pouvons désormais donner quelques exemples de transformations de la fonction d'accès ou beating.

a - Opérateur prend :

$\bar{2}\ \bar{2}\ \uparrow$ M donne une matrice de 2x2 éléments, formée des 2 dernières lignes et des 2 dernières colonnes de M, soit:

5 12
0 2

La machine naïve va demander une zone mémoire pour 4 éléments, y ranger les éléments M[0;1], M[0;2], M[1;1], M[1;2]. Le descripteur du résultat R sera alors :

rang : 2
dimensions : 2 2
vecteur de poids : 2 1
adresse : 2512

Les éléments rangés en mémoire à l'adresse 2512 :

```
 5  mémoire  2512
12     "     2513
 0     "     2514
 2     "     2515
```

La machine optimisée va simplement créer un nouveau descripteur pour le résultat R, en utilisant le **même espace mémoire** .

rang : 2
dimensions : 2 2
vecteur d'accès : 3 1
adresse : 1041

Par rapport au descripteur de M, les seules modifications concernent les dimensions et l'adresse ; elles ne dépendent que du descripteur de M et des valeurs de l'opérande gauche de ↑.

b - Transposition : ⍉ M

L'utilisateur obtiendra sur son terminal :

```
 6  19
 5   0
12   2
```

La machine d'Abrams construira le descripteur :

rang : 2
dimensions : 3 2
vecteur d'accès : 1 3
adresse : 1040

c - Restructuration : 4 ⍴M

Le descripteur du résultat optimisé sera :

rang : 1
dimension : 4
vecteur d'accès : 1
adresse : 1040

Le beating fait partie de ces rares optimisations d'espace et de temps - le gain de l'un se traduisant en général par une perte de l'autre -.

Abrams a prouvé mathématiquement la validité des transformations de beating et démontré un théorème de complétude autorisant l'application en cascade sur le même objet de plusieurs opérations de sélection. Ainsi l'expression suivante :

⍉3 3↓⌽12 8↑A[3;⍳20;]

comporte 5 sélections pour lesquelles de telles transformations sont possibles. Le temps d'évaluation d'une telle expression ne dépend pas du nombre d'éléments de la structure d'appui A, aucune demande de mémoire pour le stockage de calculs temporaires n'est nécessaire.

2°) - Le **drag along** :

C'est l'opération consistant à **différer** le plus longtemps possible les opérateurs **productifs** du langage à l'aide des opérations de sélection. Prenons par exemple l'expression devenue classique :

3 ↑ A + B (1)

dans laquelle A et B sont chacuns des vecteurs de 1000 éléments.

Une machine naïve va calculer A+B, donc engendrer 1000 additions élémentaires, 1000 rangements en mémoire temporaire, puis prendre les trois premiers éléments comme résultat.

Le drag along consiste à appliquer **d'abord** l'opération "prend" aux objets A et B, en modifiant leur descripteur, ce qui rend deux objets virtuels A' et B', puis exécuter A'+B', ce qui ne demande que trois additions et élimine le besoin en mémoire temporaire de 1000 valeurs. Abrams dit qu'il a différé l'opération "+".

En fait le drag along apparaît comme une méthode de transformation du **contenu sémantique** d'une expression APL. Dans l'exemple proposé, nous voyons que la machine naïve effectuerait un travail grossièrement équivalent en évaluant l'expression :

(3 ↑ A) + (3 ↑ B) (2)

Abrams lui-même mentionne les désavantages que peuvent

entraîner l'emploi de cette méthode. Nous connaissons bien ces problèmes par l'emploi des compilateurs qui transforment un programme rédigé dans une langage donné et ayant un contenu sémantique précis, en un autre langage - machine ou intermédiaire - ayant inéluctablement un contenu sémantique différent. Pour nous en convaincre, comparons les deux expressions :

1 ↑ 1 2 ÷ 3 0 et (1 ↑ 1 2) ÷ (1 ↑ 3 0)

L'erreur de domaine aura "échappé" à la machine optimisée. Est-ce cependant vraiment important ? Remarquons en outre qu'il n'est pas toujours possible de trouver des équivalents sémantiques comme (1) et (2) dans le cas de toutes les opérations de drag along.

IV.3 - Essais de compilation

Depuis la thèse d'Abrams proposant un code d'ordre pour une machine orientée APL, un certain nombre d'essais ont été tentés dans le but de générer du code exécutable directement sur la machine hôte à partir des lignes source d'APL.

Deux voies ont été ouvertes :

- compilation statique de sous-ensemble d'APL
- compilation dynamique et réutilisation de code machine.

IV.3.1 - Compilation statique

Elle s'applique à un sous-ensemble du langage dans lequel les objets ont des attributs fixes : une variable définie comme une matrice numérique entière ne peut rester qu'une matrice numérique entière, ses seules valeurs étant modifiables ; une fonction définie a une syntaxe immuable, les attributs de ses paramètres formels ou de son résultat ne pouvant évoluer. Le problème de la liaison dynamique des noms aux objets disparaît.

De telles méthodes sont proposées dans (I13,I14,I15,I16). Ces restrictions correspondent pratiquement à l'utilisation d'APL la plus courante dans laquelle, pour des raisons de clarté de programmation et de maintenance, ou tout simplement parce que la mise au point des programmes est terminée, les attributs des objets ne varient plus.

Le langage acquiert alors des caractéristiques voisines de celles d'ALGOL 60 et devient ainsi compilable. En fait la majorité

des opérateurs APL sont traduits par des appels à des sous-programmes système et non par la génération effective du code exécutable équivalent qui serait très certainement trop vaste. Des essais sur un matériel adapté à ALGOL 60 ont montré la possibilité d'une telle tentative. Aucune conclusion générale n'a été jusqu'alors tirée et aucun constructeur ne commercialise une telle réalisation.

IV.3.2 - Compilation dynamique

Cette méthode est relativement plus originale et ambitieuse, puisque plus aucune restriction n'est faite sur le langage à compiler. A chaque ligne source d'une fonction APL peuvent être attachés un segment de code machine exécutable et des tables de validation. Ce code est généré au cours de la première exécution effective de la ligne et pari est fait qu'il pourra resservir aux exécutions suivantes de la ligne.

Avant chaque nouvelle exécution de la ligne, on testera dès lors si le contexte d'appel n'a pas changé, et l'on pourra exécuter à nouveau le même code. Le contexte renferme des renseignements sur les noms utilisés dans la ligne :

- nature, type, rang et dimensions pour les variables
- syntaxe, existence d'un résultat pour les fonctions

et peut être testé avant chaque exécution de la ligne, ou encore après chaque retour de fonction appelée dans celle-ci (à cause des effets de bord).

Code et contexte sont détruits en cas de modification de la ligne.

Le système APL/3000 de Hewlett-Packard, décrit dans (I17), offre entre autres particularités intéressantes un compilateur incrémental adapté à APL.

Son fonctionnement de principe est le suivant :

Une fonction utilisateur est conservée sous une forme compactée, dite "S-code" (cf II.3.1).

Lors de l'exécution de la fonction, une analyse de chaque ligne est faite, générant un ou plusieurs arbres syntaxiques, dits "D-trees" - le nombre d'arbres est égal au nombre d'opérations d'affectation contenues dans la ligne -. Ces arbres, après diverses optimisations du type beating et drag along, deviennent l'entrée d'un générateur de code exécutable qui produit un fragment de code, dit

"E-code" associé à sa signature. Cette signature permettra de déterminer par la suite si les données référencées par le "E-code" n'ont pas changé.

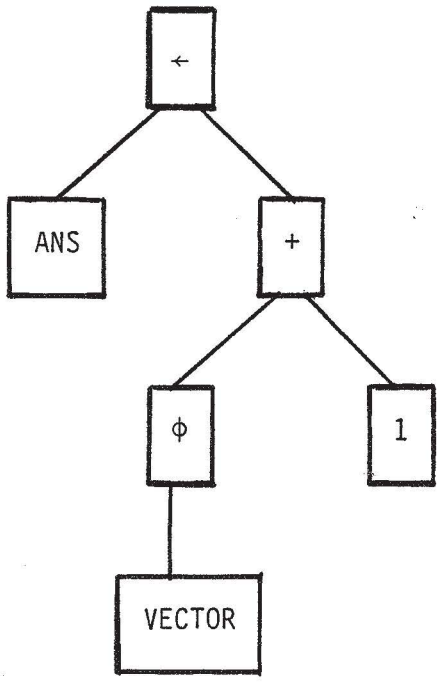
Après la première exécution de la ligne ou du fragment de ligne correspondant au "E-code" généré, celui-ci reste attaché, ainsi que la signature à la fonction APL.

Au cours des exécutions ultérieures de cette fonction, la signature permet de savoir si le code généré est toujours adapté aux données à traiter : il peut alors être directement réutilisé. Dans le cas contraire, un nouveau code est généré, qui est cette fois choisi moins dépendant des données : natures et structures restent figées, mais le code ne dépend pas des dimensions des objets. Afin de pouvoir distinguer les deux types de code, ils sont dits "Hard" ou "Soft".

Voici un exemple de tels codes générés pour une instruction APL très simple :

ANS←(⍬*VECTOR*)+1

VECTOR est un vecteur entier de 3 éléments. L'analyse syntaxique ne détecte qu'un seul "D-tree" :



Arbre syntaxique

Les instructions sont générées pour une machine à pile. Nous utilisons un langage de description algorithmique proche d'ALGOL.

a) Algorithme du "E-code hard":

```
Début
    TEMP:=obtenir-mémoire(3);
    INDEX-ECRITURE:=0; INDEX-LECTURE:=2; LIMITE:=2;
    tantque INDEX-ECRITURE ≠ LIMITE faire
        empiler(1); empiler(VECTOR[INDEX-LECTURE]);
        ajouter;
        TEMP[INDEX-ECRITURE]:=dépiler;
        incrémenter INDEX-ECRITURE;
        décrémenter INDEX-LECTURE;
    fintant
    affecter(ANS,TEMP);
fin.
```

b) Algorithme du "E-code soft":

```
Début
    TEMP:=obtenir-mémoire(ELCNT);
    INDEX-ECRITURE:=0;
    INDEX-LECTURE:=OFFSET+(RHO-1)xDEL;
    INCREMENT-LECTURE:=-DEL;
    LIMITE:=RHO;
    tantque INDEX-ECRITURE ≠ LIMITE faire
        empiler(1); empiler(VECTOR[INDEX-LECTURE]);
        ajouter;
        TEMP[INDEX-ECRITURE]:=dépiler;
        incrémenter INDEX-ECRITURE;
        INDEX-LECTURE:=INDEX-LECTURE+INCREMENT-LECTURE;
    fintant
    affecter(ANS,TEMP);
fin.
```

Dans ce cas précis, le code mou est indépendant de la taille de VECTOR (qui doit être de 3 pour réutilisation du code dur)

et de son implantation dans la mémoire - les éléments de VECTOR étant supposés rangés de façon contiguë et par ordre croissant d'indice dans le cas du code dur -.

La réutilisation de code machine donne parfois des résultats intéressants. Ainsi, la première exécution d'une certaine fonction de test demande 240 ms de temps CPU, alors que les utilisations ultérieures, sautant la phase de compilation ne prennent que 20 ms.

Différents tests effectués sur APL/3000, ainsi que les conclusions de deux études comparatives de différents systèmes APL montrent cependant que les performances mesurées ne semblent pas à la hauteur des résultats escomptés : par exemple des rapports de 1 à 2 sont souvent obtenus sur les temps CPU de certaines fonctions de test entre APL/3000 et APL/16, dont l'unité centrale est cependant moins puissante.

PARTIE II

les OPTIMISATIONS

Cette deuxième partie présente les optimisations qui ont été choisies dans la réalisation.

Le troisième chapitre introduit une définition très générale des j-vecteurs, dresse la liste des opérations conservant cette structure et étend à cette définition plus large les règles du beating proposées par Abrams. Il présente enfin comment ont été réalisés les partages de données et plus généralement expose le principe hiérarchique de la mémoire APL et la conception de la gestion récursive des blocs-pointeurs. Cette gestion automatique de la cohérence mémoire permettra une intégration aisée des tableaux généralisés de la troisième partie.

Le quatrième chapitre s'intéresse plus spécialement au point sensible de tout système APL : les accès aux objets. Cette étude menée sous trois éclairages différents : celui des opérateurs, celui des variables optimisées et celui de la mémoire virtuelle simulée, permet de dégager de nouvelles optimisations, d'unifier les fonctions d'accès aux variables et conduit à la définition d'un compilateur dynamique d'accès : le **générateur de liens** .

C H A P I T R E 3

I - LES J-VECTEURS

Il est possible de concevoir une représentation interne particulièrement bien adaptée au langage : le type **j-vecteur** ou progression arithmétique, sous la forme d'un triplet de 3 nombres n, b, p. Ces trois valeurs représentent un vecteur V de n éléments, tel que :

$\forall i \in [0,n[$, V [i] ait la valeur b+pxi.

Une telle représentation a été introduite par Abrams (I6) et effectivement utilisée dans certaines réalisations (R5,I5,I17).

En fait, on ne cherche pas à reconnaître si une donnée APL est un j-vecteur, mais on tient compte du fait que ces derniers sont générés par certains opérateurs et conservés par d'autres.

Cette optimisation entraîne des gains de mémoire importants et des économies de temps de traitement dans tous les opérateurs laissant le type j-vecteur invariant. Le retour à une représentation interne plus classique doit être prévu dans la réalisation, comme l'illustre l'exemple suivant:

```
V←13+5×⍳1000
V[3]←.1
```

I.1 - Définition

Par définition, nous appellerons j-vecteur toute composition affine de (⍳N), ou encore nous dirons que V↔j(n,b,p) représente la progression arithmétique finie de n éléments, de raison p et d'origine b. Nous appellerons p le pas et b la base du j-vecteur. Si n est nul, V est alors le vecteur vide.

$$\forall n \in \mathbf{N} , \forall b,p \in R , \underline{j}(n,b,p) \leftrightarrow b+p \times \iota n$$

I.2 - Extension des scalaires

Il est intéressant d'étendre symboliquement un scalaire entier à un j-vecteur d'éléments égaux, donc de pas nul.

$$\forall n \in \mathbf{N} , \forall s \in R , s \; E_n \; \underline{j}(n,s,0)$$

Ce choix d'une raison nulle permet de simplifier et de régulariser les opérations optimisées entre scalaires et j-vecteurs.

I.3 - Opérations sur les j-vecteurs

Nous définirons tout d'abord les ensembles J_n de j-vecteurs de n éléments : $\forall n \in N$, $J_n=\{j | j=\underline{j}(n,b,p), b,p \in R\}$

1°) - Opérations arithmétiques :

a - L' **identité** , le **plafond** , le **plancher** laissent les entiers invariants, ce sont donc des identités sur tout j-vecteur entier ($b,p \in Z$).

b - **Opposé** d'un j-vecteur:

$\forall j \in J_n$, tel que $j=\underline{j}(n,b,p)$, $-j \in J_n$ et $-j=\underline{j}(n,-b,-p)$.

c - **Addition** de 2 j-vecteurs : c'est une loi de composition interne dans tout J_n :

$\forall j_1,j_2 \in J_n$, tels que $j_1=\underline{j}(n,b_1,p_1)$ et $j_2=\underline{j}(n,b_2,p_2)$,

$j_1+j_2 \in J_n$ et $j_1+j_2=j(n,b_1+b_2,p_1+p_2)$.

$\forall s \in R$, $E_n(s)=\underline{j}(n,s,0)$, et $\forall j \in J_n$, tel que $j=\underline{j}(n,b,p)$,

$s+j \leftrightarrow j+s \leftrightarrow j+E_n(s) \leftrightarrow \underline{j}(n,b+s,p)$.

d - **Soustraction** de 2 j-vecteurs : c'est aussi une loi interne.

$\forall j_1,j_2 \in J_n$, tels que $j_1=\underline{j}(n,b_1,p_1)$ et $j_2=\underline{j}(n,b_2,p_2)$,

$j_1-j_2 \in J_n$ et $j_1-j_2=\underline{j}(n,b_1-b_2,p_1-p_2)$.

$\forall s \in R$, et $\forall j \in J_n$, tel que $j=\underline{j}(n,b,p)$,

$s-j \leftrightarrow \underline{j}(n,s-b,p)$ et $j-s \leftrightarrow \underline{j}(n,b-s,p)$.

e - **Multiplication** de 2 j-vecteurs : soient j_1 et j_2 deux éléments de J_n, tels que $j_1=\underline{j}(n,b_1,p_1)$ et $j_2=\underline{j}(n,b_2,p_2)$. Evaluons leur produit :

$$\forall i \in [0,n[\ , \ (j_1 \times j_2)[i] = (b_1+p_1 \times i)(b_2+p_2 \times i)$$
$$= b_1b_2+i \times (b_2p_1+b_1p_2)+i^2p_1p_2$$

Ce n'est un j-vecteur que si le produit p_1p_2 est nul, ce qui veut dire que l'un des deux opérandes au moins est de raison nulle (ou scalaire étendu). La multiplication de tout j-vecteur par un scalaire est une loi externe sur tout J_n.

$\forall s \in R$, $\forall j \in J_n$, $j=\underline{j}(n,b,p)$, $s \times j \in J_n$, $s \times j = \underline{j}(n,bs,ps)$.

2°) - Opérations de sélection :

Les règles du beating sont applicables aux j-vecteurs. Il suffit de poser pour $j=\underline{j}(n,b,p)$:

abase := b et wv := p

et d'appliquer les règles exposées au § II.1 . Les opérateurs $\uparrow 2, \downarrow 2, \phi 1, \phi 12$ n'affectent pas la représentation j-vecteur.

Exemples :

$\bar{4} \uparrow 2x \iota 10 \leftrightarrow \bar{4} \uparrow \underline{j}(10,0,2) \leftrightarrow \underline{j}(4,0+2x(10-4),2) \leftrightarrow \underline{j}(4,12,2)$

$\phi\ 3+2x \iota 5 \leftrightarrow \phi \underline{j}(5,3,2) \leftrightarrow \underline{j}(5,3+2x4,\bar{2}) \leftrightarrow \underline{j}(5,11,\bar{2})$

Indexation d'un j-vecteur par un j-vecteur : soient deux j-vecteurs $j_1 \in J_n$ et $j_2 \in J_m$ tels que :

$j_1=\underline{j}(n,b_1,p_1)$ et $j_2=\underline{j}(m,b_2,p_2)$

Nous supposerons en outre que les valeurs de j_1 sont des indices valides de j_2, à savoir que :

$b_1 \in [0,m[$ et $b_1+p_1x(n-1) \in [0,m[$.

Nous pouvons écrire:

$$\forall i \in [0,n[\ ,\ (j_2\ [j_1]\ )_i = j_2\ [j_1\ [i]]$$
$$= j_2\ [b_1+p_1xi] = b_2+p_2x(b_1+p_1xi)$$
$$= (b_2+p_2xb_1)+ixp_1p_2 = j_3\ [i]$$

$j_3 \in J_n$, $j_3=\underline{j}(n,b_2+p_2xb_1,p_1p_2)$

On constate que j_3 est la composée de fonctions : $j_2 \circ j_1$.

3°) - Autres opérations :

a - **Tris** : un j-vecteur étant une suite finie monotone, le vecteur d'indices qui trie ses valeurs de façon ascendante ou descendante est également un j-vecteur. Le signe du pas du j-vecteur et le sens du tri déterminent la monotonie de la suite résultat.

cas $p \geq 0$: $\Delta\, \underline{j}(n,b,p) \leftrightarrow \underline{j}(n, \square IO,1)$

cas $p < 0$: $\Delta\, \underline{j}(n,b,p) \leftrightarrow \underline{j}(n,n-1+\square IO,\bar{1})$

b - **Restructuration** : pour peu que $(x/\rho r)=1$ et $r \leq n$, c'est à dire sans option cyclique :

$r \rho \underline{j}(n,b,p) \leftrightarrow \underline{j}(r,b,p)$

c - **Linéarisation** : trivialement la forme linéarisée d'un j-vecteur reste un j-vecteur. La linéarisation d'un scalaire est également un j-vecteur.

I.4 - Représentation interne

Quelques réalisations d'interprète APL utilisent cette représentation interne :

- Arithmetic progression vectors d'APL/3000 (I17)
- J-tableaux d'APL15 (I5).
- J-vecteurs d'APL/4004 (R5).

Les auteurs s'accordent pour les représenter par un triplet (n,b,p). Notons que la plupart des résultats précédents restent indépendants des représentations internes choisies pour représenter b et p. En général, le choix se porte vers une représentation entière ; cependant, une représentation flottante peut être employée, comme sous APL15, par exemple, qui connaît les j-tableaux en représentation simple et étendue. Toutefois, comme sur les machines actuelles le domaine de représentation des nombres entiers est limité à un sous-ensemble de Z centré en 0, il faut prévoir un mécanisme de changement de représentation.

Par exemple, sur une machine possédant une arithmétique entière 16 bits, tout j-vecteur doit vérifier :

$S = [\bar{}32678, 32767]$

$\underline{j}(n,b,p) \subset S \Rightarrow b \in S \text{ et } b+px(n-1) \in S \quad (1).$

Si la réalisation comprend les j-vecteurs en représentation flottante et si la condition (1) n'est pas vérifiée par le résultat :

$\underline{j}(n,b_1,p_1) + \underline{j}(n,b_2,p_2) \leftrightarrow \underline{j}(n,b_1+b_2,p_1+p_2)$

(entier) (entier) (flottant)

Dans le cas contraire, un changement de structure "j-vecteur → vecteur" s'impose; ce problème se pose également pour l'affectation indicée.

II.1 - Présentation des règles du beating

Dans l'essence, ces règles sont identiques à celles que l'on peut trouver dans (I6). Nous les présentons toutefois car nous les avons rendues dépendantes de l'origine et adaptées à notre définition plus générale des j-vecteurs. Les attributs de l'opérande principal de l'opération de sélection sont notés :

rang : rang
r : vecteur de dimension
abase : constante additionnelle
w : vecteur d'accès

D'autre part, nous donnons pour chaque cas la condition d'application de la règle de beating. Ces règles sont rédigées en langage APL.

1°)- **Prend :** A ↑ M

résultat
rang
|A
abase+w+.x(A<0)xr-|A
w
Condition : ∧/0≤r-|A ⍝ sans complétion

2°)- **Laisse :** A ↓ M

résultat
rang
0⌈r-|A
abase+w+.x(A>0)x|A
w
Condition : 1 ⍝ toujours possible

3°)- **Renversement :** ⌽[K] M

(K←K-⎕IO)
résultat
rang
r
abase+w[K]x(r[K]-1)
w ◇ w[K] ← -w[K]
Condition : 1 ⍝ toujours possible

4˜)- **Transpositions** : A⍉M (pour ⍉ M, A←⍉⍳⍴ M)

résultat

q←1+⌈/A-⎕IO

rho←q↑r ◇ k←0 ◇ tantque k<q faire rho[k]←⌊/(A=k)/r fin

abase

wv←q↑w ◇ k←0 ◇ tantque k<q faire wv[k]←+/(A=k)/w fin

Condition : 1 ⍝ toujours possible

5˜)- **Indexation** : M[I0;I1;...;Ip-1]

Les p index sont scalaires, j-vecteurs ou omis. Pour les attributs de chaque index I_k, on prend selon les cas :

Attributs	Scalaire s	j(n,b,p)	Omis
$rang_k$	0	1	1
r_k	⍳0	n	r[k]
$abase_k$	s	b	⎕IO
w_k	⍳0	p	1

résultat

+/rang0,rang1,...,rangp-1

,r0,r1,...,rp-1

abase+w+.x(abase0,abase1,...,abasep-1)-⎕IO

,(w[0]xw0),(w[1]xw1),...,(w[p-1]xwp-1)

Condition : ∧/(Ci I0),(Ci I1),...,(Ci Ip-1)

La fonction C_i vérifie qu'une variable est un scalaire ou une progression : $C_i \leftarrow\!\!\leftarrow C_s \wedge C_j$:

```
    ∇ Z←CS S
[1]   Z←0=⍴⍴S
    ∇

    ∇ Z←CJ J
[1]   N←⍴J ◇ B←1↑J ◇ P←--/2↑J
[2]   Z←∧/(1=⍴N),P=(1↓J)-1↓¯1⌽J
    ∇
```

6°)- **Structuration** : A ρ M

a - Restructuration compatible sans remplissage cyclique :

résultat
rang
,A
abase
w
Condition C_1 : ((ρ A)= ρρM) ∧ (x/A) ≤ x/ ρ M

b - Le vecteur d'accès de M égale son vecteur de poids naturel :

résultat
ρ A
,A
abase
1 ↓⌽ x\⌽ A,1
Condition C_2 : ∧ /w=1↓⌽ x\⌽ r,1

7°)- **Linéarisation** : ,M

résultat
1
,x/ ρ M
abase
,1
Condition : C_2

8°)- **Identités** : +M ou ⌈M ou ⌊M ou A←M

résultat
rang
r
abase
w
Conditions C_3 et C_4 : ∧/M= ⌈M et ∧/M= ⌊M

II.2 - Applications à une réalisation

II.2.1 - Conditions d'optimisation : Certaines conditions d'application paraissent très complexes. Dans la pratique d'une réalisation, elles se traduisent en fait par de simples tests sur les informations d'attribut conservées au niveau du descriptif d'une donnée :

C_s <==> $RANG_1=0$

C_j <==> $OPT_1=\bar{}1$ (représentation j-vecteur)

C_1 <==> $ELCNT_1=RANG_2$ et $ELCNT_r \leq ELCNT_2$

C_2 <==> $OPT_2=0$ (variable non optimisée)

C_3,C_4 <==> $1< TO_2 \leq 3$ (type entier ou booléen)

Les indices 1,2 et r correspondent aux trois objets régulièrement en présence dans une opération APL :

$OBJET_r \leftarrow OBJET_1 \circledcirc OBJET_2$

II.2.2 - Résultat d'une optimisation : dans le même esprit, à l'issue de toute optimisation de beating, on s'efforcera d'identifier le résultat produit afin de profiter au mieux de certaines représentations simplifiées de variable :

- ($RANG_r=0$) : le résultat est scalaire
- ($ELCNT_r=0$) : le résultat est vide (partager les valeurs de l'opérande droit n'a pas de sens)
- ($OPT_2=\bar{}1$) : l'optimisation s'applique à un j-vecteur, le résultat est donc un j-vecteur.

Dans le cas général, l'optimisation a porté sur un opérande droit dont les valeurs sont rangées dans un bloc de mémoire. Elle a permis de sélectionner un sous-tableau rectangulaire défini par les attributs ($RANG_r$,$RHOR_r$) et dont les éléments seront obtenus par l'emploi de la fonction d'accès de paramètres ($ABASE_r$,WV_r) sur le bloc des valeurs de l'opérande droit. Il y a alors partage de données entre opérande et résultat.

II.2.3 - Intérêt du beating : on peut se demander si le beating présente un grand intérêt quand la taille du sous-tableau sélectionné reste nettement inférieure à celle de la donnée d'appui. Définissons tout d'abord, pour les optimisations présentées, le **taux de partage** comme le rapport des nombres d'éléments du résultat et de l'opérande droit :

$$\Theta = \frac{ELCNT_r}{ELCNT_2}$$

On voit par exemple que dans le cas des identités ou des restructurations complètes ce taux est optimal ($\Theta=100\%$).

a) Dans tous les cas où les opérations de beating créent

des résultats **temporaires** , on a intérêt à l'appliquer, quel que soit les taux de partage rencontrés. Par exemple, dans l'expression :

```
        +1 1⌽3 3↓⌽12 8↑A[3;⍳20;]
  535  544  553  562  571                    ⍝  A←3 25 10⍴⍳750
```

b) En revanche, lorsque le résultat est **affecté** , et que la variable d'appui est temporaire, le partage paraît critiquable, dans la mesure où il introduit une indirection mémoire et maintient actif un bloc peut-être plus important en taille que sa valeur d'emploi. Par exemple :

```
A←30↑1000⍴1    ⍝  FAUX PARTAGE

A←30↑B←1000⍴1  ⍝  VRAI PARTAGE
```

La politique que nous proposons alors consiste à utiliser le taux de partage comme discriminant : pour $\Theta \geq \Theta_m$ (fixé), on acceptera le partage, dans le cas contraire, on "désoptimisera" après coup - ce qui évite d'avoir à écrire les opérateurs de sélection en double exemplaire.

III - LA MEMOIRE PARTAGEE

Cette troisième optimisation a pour but général d'éviter une trop grande déperdition de mémoire, APL étant gros consommateur de cette ressource.

Nous avons donc décidé de gérer les allocations et libérations de la mémoire virtuelle à deux niveaux. Le premier ne s'intéresse qu'à une gestion classique de blocs de taille variable de la mémoire, le second se consacre aux différents partages pouvant être réalisés sur ces blocs et, plus généralement à la gestion de la cohérence de ces partages pour les blocs contenant des pointeurs sur d'autres blocs.

Au deuxième niveau, la mémoire occupée apparaît alors comme un ensemble de blocs structurés sous la forme d'un **graphe orienté sans circuit** .

III.1 - Hiérarchie de la mémoire

Tous les objets manipulés par l'interprète sont définis par une adresse universelle mémoire (G5). Un bloc mémoire peut contenir des informations et, en particulier, d'autres adresses de blocs.

Le partage de la même information est rendu possible en admettant l'existence de plusieurs pointeurs sur un même bloc. Il est alors nécessaire d'associer à chaque bloc un **compteur d'utilisation** , (u), indiquant à combien de "propriétaires" appartient ce bloc. Un tel bloc ne peut être effectivement détruit (rendu à l'allocateur du premier niveau) que lorsque le compteur associé devient nul.

On obtient ainsi pour la gestion de la mémoire la hiérarchie présentée à la fig. 3.

III.1.1 - Les primitives de la G.D.M. : le gestionnaire de mémoire du premier niveau n'est sollicité que pour allouer ou libérer **physiquement** des morceaux de mémoire virtuelle contigus (les grains). Deux primitives sont utilisées (cf. chap. 2) :

```
allocation :   A := MGET(n);
libération :   MKILL(A);
```

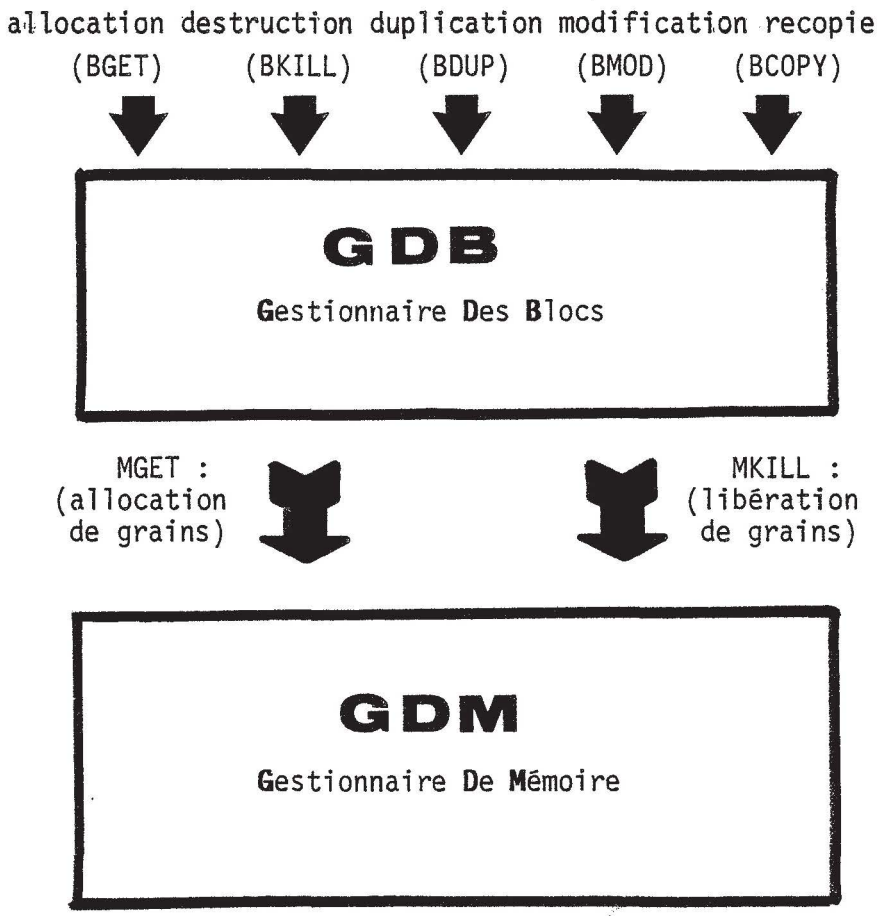


Figure 3 : La hiérarchie de la mémoire

- n est le nombre de grains de mémoire virtuelle désirés

- A est l' **adresse virtuelle** du bloc obtenu. Elle est codée sur deux mots machine et définit un triplet :

A :=: (p,l,d)

. p : désignation universelle de page (type d'espace + numéro de page dans l'espace (G5))

. l : taille moins une unité du bloc en grains

. d : déplacement relatif du premier grain du bloc dans la page.

Une adresse virtuelle contient donc les informations décrivant la catégorie de mémoire utilisée, la localisation du bloc dans cet espace et la **taille** du bloc en unités d'allocation.

III.1.2 - Les primitives de la G.D.B. : le gestionnaire de blocs alloue et libère **logiquement** les blocs de mémoire. Ses primitives sont :

1°) - **Création** d'un bloc :

A := **BGET** (elcnt,to);

- elcnt : nombre d'éléments désirés dans la représentation
- to : type de la représentation (booléen,entier,etc.)
- A : adresse virtuelle du bloc obtenu (cf III.1.1).

Cette primitive se traduit par une demande d'allocation au gestionnaire de mémoire du nombre de grains nécessaires, en tenant compte des paramètres elcnt et to et des informations de contrôle qui seront mises en fin de bloc. Le résultat est un bloc dont le compteur d'utilisation est initialisé à 1.

2°) - **Destruction** d'un bloc :

A := **BKILL** (A);

Le gestionnaire décrémente le compteur associé au bloc. S'il devient nul, le bloc est rendu à la mémoire libre par un appel de MKILL.

3°) - **Duplication** d'un bloc :

B := **BDUP** (A);

Cette primitive représente une demande de partage du bloc

A. Le compteur du bloc associé est simplement incrémenté. L'adresse rendue est alors la même (B $\underline{=}$ A). Si ce compteur déborde, ce qu'il faut bien prévoir ($u \leq 32767$, dans notre réalisation), une copie physique, d'adresse B, est créée par un appel de MGET.

4°) - Demande de **modification** d'un bloc :

B := **BMOD** (A);

Cette primitive est une demande de modification d'un bloc. Cette requête n'est honorée que si le compteur du bloc associé (u_A) vaut 1, auquel cas l'adresse rendue est A.

Dans le cas contraire, une copie physique est créée, d'adresse B, et le compteur du bloc A est décrémenté. On isole ainsi une image individuelle du bloc A.

5°) - Demande de **recopie physique** d'un bloc :

B := **BCOPY** (A);

Cette primitive se traduit par une demande d'allocation de mémoire brute (MGET) et par la recopie "mot à mot" du contenu du bloc source (A) sur le bloc cible (B). Le compteur associé de ce dernier est initialisé à 1. Cette primitive peut être appelée par BDUP ou BMOD.

III.2 - Structure des blocs-pointeurs

Le problème se complique si l'on admet maintenant qu'un bloc puisse contenir des pointeurs sur d'autres blocs : il faut en effet pouvoir gérer les blocs pointés par le bloc pointeur en cas de destruction ou de recopie physique de ce bloc. A cette fin, nous ajoutons donc deux autres informations de contrôle caractéristiques d'un bloc :

- le type du bloc (t)
- le nombre de pointeurs que contient le bloc (p).

Une analyse détaillée des différentes structures de données manipulées par l'interprète, les variables, les fonctions et ses propres tables de travail, nous a permis de dégager 4 grandes catégories de bloc, chaque catégorie correspondant à l'organisation interne des pointeurs dans ces blocs :

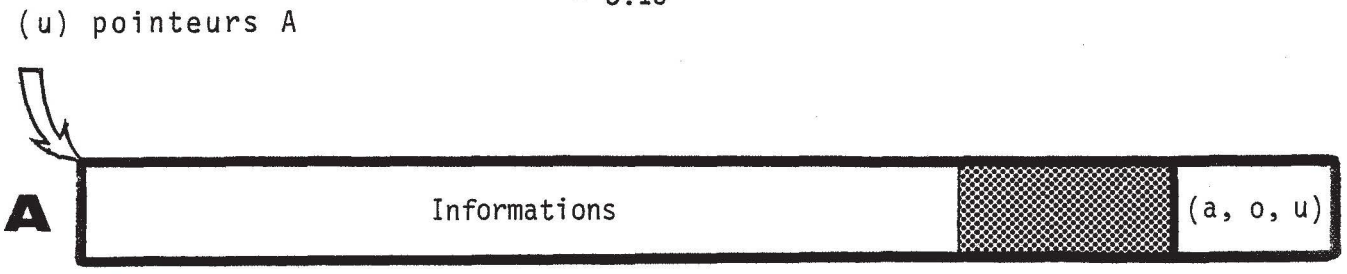


Figure 4.1: Bloc sans pointeur (t=a, p=0)

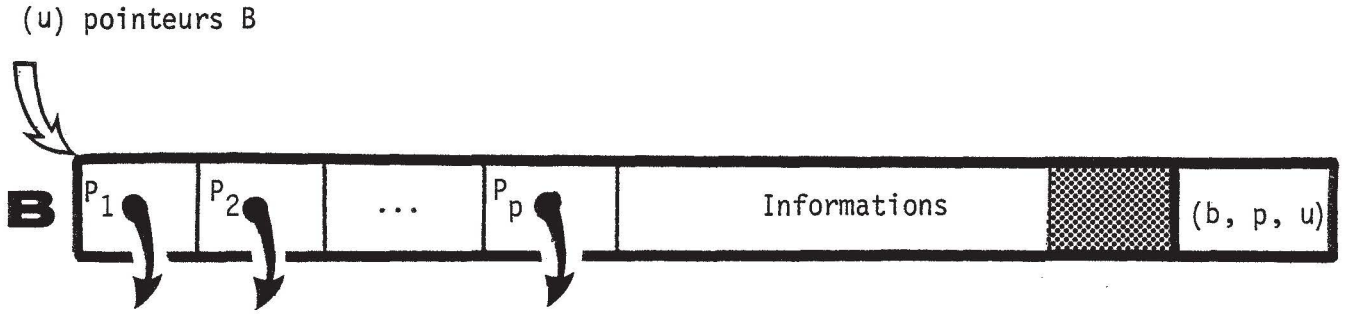


Figure 4.2: Bloc - liste de pointeurs (t=b, p pointeurs)

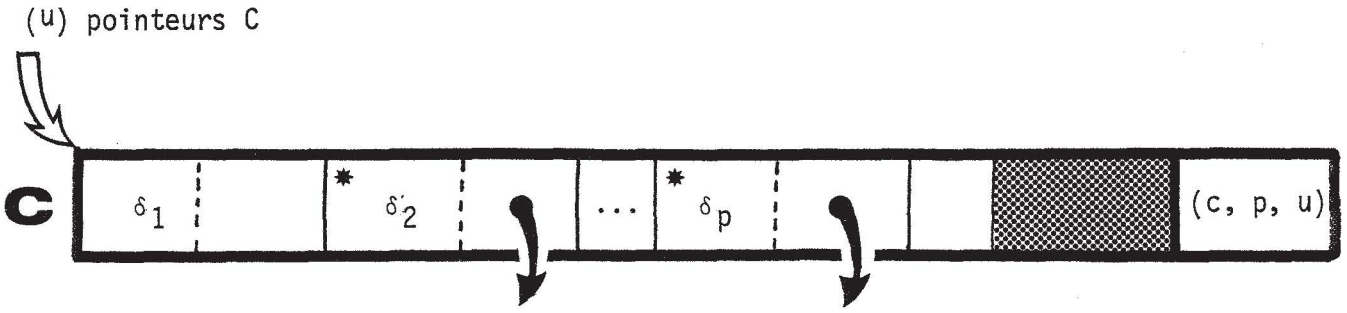


Figure 4.3: Bloc - liste de descripteurs (t=c, p descripteurs, pas forcément p pointeurs)

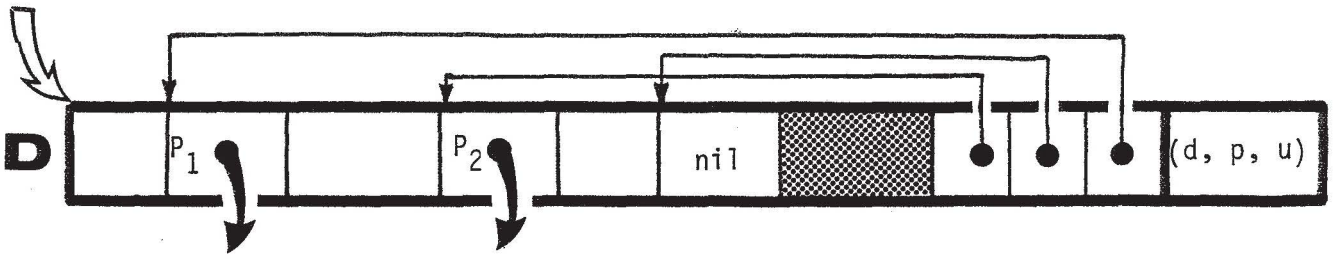


Figure 4.4 : Bloc contenant (p) pointeurs aléatoirement répartis

Les p antépénultièmes informations contiennent des pointeurs relatifs sur les adresses contenues dans le bloc

a - Bloc **sans pointeur** (t=a) : ils seront utilisés pour les variables simples (ni généralisées, ni optimisées), la table des symboles, etc. Leur structure est représentée sur la fig. 4.1.

b - Bloc **liste de pointeurs** contigus (t=b) : dans de tels blocs sont rangées (p) adresses virtuelles de façon contiguë depuis le début du bloc (fig.4.2). Le reste du bloc peut contenir des informations de type (a). Ils correspondent typiquement à des tables d'adresses.

c - Bloc **liste de descripteurs** contigus (t=c) : l'organisation est la même que précédemment, à la différence qu'un descripteur ne contiendra pas forcément un pointeur.

Un descripteur contient les informations d'attributs d'une variable les plus fréquemment utilisées : rang, type de la représentation, etc., et éventuellement un pointeur sur un bloc mémoire contenant le complément de description et les valeurs. En fait, un indicateur, contenu dans tout descripteur, permet de reconnaître un descripteur contenant un pointeur d'un descripteur qui n'en contient pas (fig.4.3).

La table des descripteurs de la zone active, les blocs principaux des fonctions définies (G9), les tableaux généralisés en sont des exemples.

d - Bloc contenant des **pointeurs aléatoirement répartis** (t=d) : la répartition des (p) pointeurs est alors précisée par une liste auxiliaire, rangée en fin de bloc, des (p) positions relatives de ces pointeurs par rapport au début du bloc (fig.4.4). Ce type correspond à toutes les structurations de pointeurs ne rentrant pas dans les trois catégories précédentes. Par exemple, le descriptif d'une variable optimisée contient un pointeur sur un bloc de valeurs.

Tout bloc est donc caractérisé par un triplet (t,p,u), ce qui permet au Gestionnaire De la mémoire des Blocs de savoir :

- combien de fois le bloc est pointé (u)
- si le bloc contient des pointeurs ($t \neq a$)
- auquel cas, combien il en contient (p)
- et comment ces pointeurs sont arrangés dans le bloc (t).

III.3 - Application récursive d'une primitive sur un bloc-pointeur

Compte tenu de la grande variété d'organisation des blocs-pointeurs et pour rendre la gestion des blocs entièrement autonome, nous avons créé une nouvelle primitive, que nous notons BEXFCT ou parfois ⊛ , permettant d'appliquer une primitive donnée sur toute la descendance d'un bloc, c'est à dire sur tous les blocs pointés. Cette primitive étant récursive, nous avons dû intégrer au noyau un service logiciel permettant la récursivité.

III.4 - Algorithmes définitifs

Les algorithmes correspondant à chaque primitive d'appel du gestionnaire des blocs deviennent (Algorithmes 1,2,3,4,5) :

```
Procédure B:=BDUP(A);
   B:=A; retoursi A=nil ; incr u_A;
   si débordement alors
      decr u_A; B:=BCOPY(A);
   finsi
finproc
```

Algorithme 1 : Duplication du bloc A.

```
Procédure B:=BKILL(A);
   B:=nil; retoursi A=nil ; decr u_A;
   si u_A=0 alors
      si t_A≠ a alors
         BEXFCT(BKILL,A);
      finsi
      MKILL(A);
   finsi
finproc
```

Algorithme 2 : Destruction du bloc A.

```
Procédure B:=BCOPY(A);
   B:=MGET(l_A); retoursi A=nil; transférer-contenu(A,B);
   t_B:=t_A; p_B:=p_A; u_B:=1;
   si t_B ≠a alors
      BEXFCT(BDUP,B);
   finsi
finproc
```

Algorithme 3 : Recopie physique du bloc A.

```
Procedure BEXFCT(fonction,A);
   entier ifils, dep; pointeur F,G; descripteur D;
   retoursi A=nil; ifils:=0;
   tantque ifils <p_A faire
      selon t_A faire
         a : erreur-système;
         b : F:=charger-pointeur(A,ifils); dep:=ifils;
         c : D:=charger-descripteur(A,ifils);
             si D <0 alors
                dep:=2*ifils+1;
                F:=charger-pointeur(A,dep);
             sinon
                F,dep:=nil;
             finsi
         d : dep:=charger-mot(A,-ifils);
             F:=charger-pointeur(A,dep);
      finselon
      G:=fonction(F);
      si dep ≠ nil alors
         ranger-pointeur(G,A,dep);
      finsi
      incr ifils;
   fintant
finproc
```

Algorithme 4 : Application récursive d'une fonction (BDUP ou BKILL) sur la descendance du bloc A .

```
Procedure B:=BMOD(A);
   B:=A; retoursi A=nil;
   si u_A ≠ 1 alors
      decr u_A; B:=BCOPY(A);
   finsi
finproc
```

Algorithme 5 : Demande de modification du bloc A.

IV - APPLICATIONS

Nous donnons quelques exemples tirés de la réalisation d'utilisation des optimisations précédentes. Pour cela nous présentons les choix de représentation des variables, quelques exemples de partages mémoire engendrés par l'exécution de certains opérateurs et le mécanisme de passage de paramètres des fonctions définies.

IV.1 - Représentation des variables

Qu'une variable soit permanente, c'est à dire associée à un **nom** , A par exemple, ou temporaire, c'est à dire correspondant seulement à une constante ou au résultat d'un calcul, ses attributs sont représentés par :

- un **descripteur primaire** , noté $\delta(A)$ et situé dans la table des descripteurs (TD) si la variable est permanente, noté ξ_i et situé dans la pile des descripteurs temporaires (TEMP) de la ligne d'exécution si la variable est temporaire.
- un éventuel **descriptif complémentaire** , noté Δ et situé en queue du bloc mémoire des valeurs de la variable ; le descripteur primaire contient alors un pointeur mémoire.

Le descripteur primaire contient les informations dites "de première nécessité" : il précise que l'objet est une variable, indique si cette variable est un scalaire (Sc), un vecteur (V), une matrice (Ma), un j-vecteur (Jv), fournit le type choisi pour la représentation des valeurs de la variable : caractère (C), booléen (B), entier (E), flottant (R), structure (St), etc.

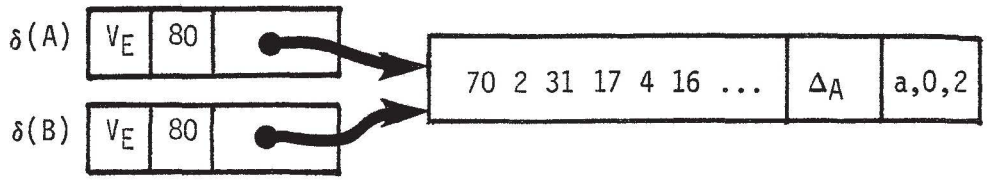
Le descriptif complémentaire contient le reste des attri-

buts de la variable (constante Abase, indicateur d'optimisation, vecteur de dimensions, vecteur d'accès, adresse d'appui, etc.).

IV.2 - Optimisations des opérateurs

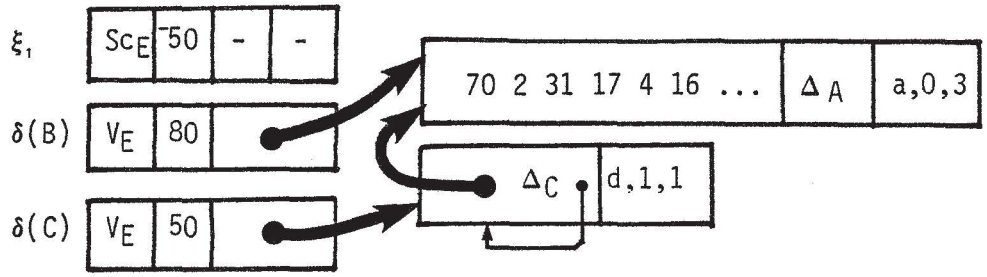
a) - Partage complet de la description et des valeurs : Dans les identités, le traitement consiste à recopier le descripteur et éventuellement à dupliquer le bloc-valeurs :

B ← ⌈ A ← 80 ? 80

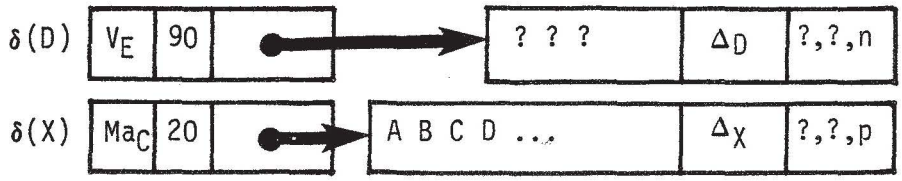


b) - Partage des valeurs seules : nous procédons à la création d'un bloc descriptif résultat (bloc mémoire de type d) qui s'appuie sur le bloc de la variable opérande. Ce dernier est dupliqué :

C ← ¯50 ↑ B

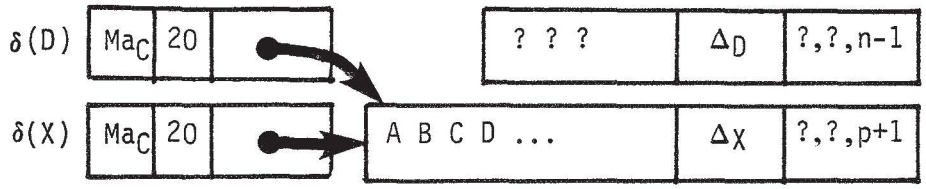


c) - Affectation : nous distinguerons l'opérande droit - l'affectant - de l'opérande gauche - l'affecté :



D ← X

On duplique d'abord le bloc de l'affectant, on détruit ensuite celui de l'affecté :



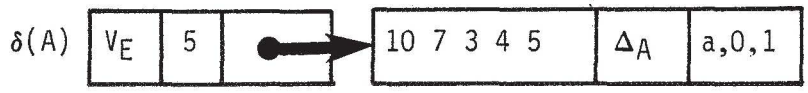
IV.3 - Passage des paramètres d'une fonction

A l'appel d'une fonction, il doit y avoir protection des paramètres passés à la fonction contre les modifications que cette dernière pourrait leur apporter : les passages de paramètres se font par valeur. La méthode générale employée par de nombreux systèmes consiste à passer un copie physique des paramètres lors de l'appel. Compte tenu de la taille des objets, les mouvements de mémoire conséquents à cette opération peuvent être très importants et surtout s'avèrent dans la majeure partie des cas **inutiles** , car, s'il peut arriver qu'une fonction modifie ses paramètres d'appel, ceci ne signifie pas que toutes les fonctions le font.

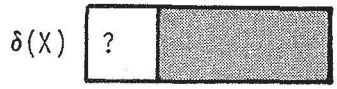
La méthode que nous employons n'entraînent de tels transferts que pour les fonctions tentant réellement de modifier les valeurs des paramètres d'entrée. Pour cela, à l'appel d'une fonction, nous sauvegardons uniquement le contenu des descripteurs de tous les objets globaux en correspondance de nom avec les objets locaux à la fonction appelée et nous dupliquons logiquement les blocs valeurs des variables paramètres, seulement lorsqu'elles sont **permanentes** .

Le principe utilisé est donc l'opposé du premier cité puisque, avant d'appeler une fonction, on suppose a priori que cette dernière ne modifiera pas ses paramètres d'appel.

```
      A←10 7 3 4 5
```

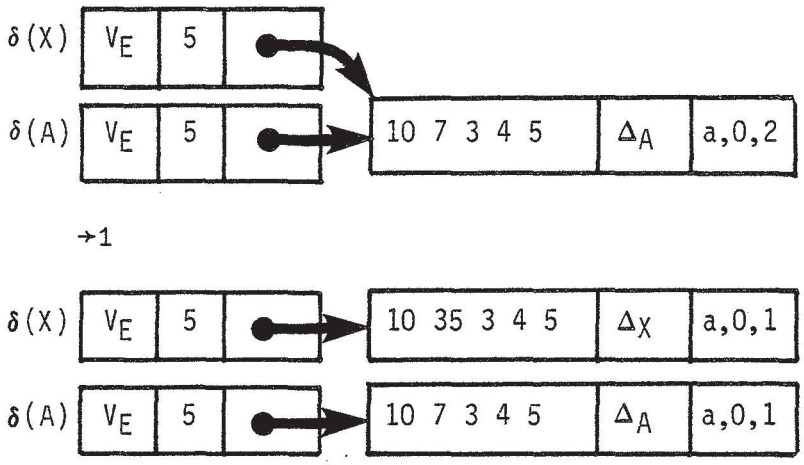



```
      ∇F X
[1]   X[2]←35∇

      1 □STOP 'F'
1

```



(Objet indéfini)

```
      F A
F[1]    *
```



L'affectation indicée ayant effectué une primitive "BMOD" sur le bloc de X afin de pouvoir le modifier, un transfert automatique a séparé les valeurs de X de celles de A. Au retour, il y a destruction logique des blocs-valeurs des paramètres et le descripteur de X est restauré (X redevient indéfini). Dans le cas présent le bloc de X est donc rendu à la mémoire libre.

Lorsque la fonction appelée ne modifie pas ses paramètres, les compteurs d'utilisation des blocs-valeurs de ceux-ci sont donc incrémentés à l'appel et décrémentés au retour : le passage des paramètres par valeur devient aussi performant qu'un passage par référence, avec les désagréables effets de bord en moins.

CHAPITRE 4

INTRODUCTION

Nous étudions de très près dans ce chapitre les accès aux variables APL lors de l'exécution des opérateurs.

Hormis l'affectation indicée, qui relève d'un cas vraiment particulier, le résultat d'une opération peut s'obtenir de trois façons différentes :

a) par la construction d'un simple descripteur : on crée un scalaire, un j-vecteur, etc.

b) par un partage des valeurs de l'opérande droit : **direct** si les valeurs restent accessibles dans l'ordre lexicographique, **indirect** si un vecteur d'accès doit être utilisé.

c) par la demande et le remplissage d'un nouveau bloc de mémoire : les valeurs utilisées pour ce remplissage résultent alors d'un traitement sur les valeurs du ou des opérandes.

Dans le dernier cas, celui qui nous intéresse, une grande partie des objets doit être accédée élément par élément. En éclairant cette étude d'accès sous trois lumières différentes : celle des opérateurs, celle des variables optimisées et celle de la mémoire virtuelle paginée, nous dégageons plusieurs optimisations de ces accès.

Cette étude débouche plus généralement sur l'introduction de la notion de "lien sur un objet APL" et sur la définition d'un compilateur dynamique de sous-programme d'accès aux objets APL.

I - ETUDE DES ACCES

I.1 - Influence des opérateurs

I.1.1 - L'adressage fonctionnel simple

Soit à évaluer l'expression, R ← A+B. Plaçons nous dans le cas plus simple où les blocs des valeurs de A, B et R sont implantés dans une mémoire uniforme avec des représentations identiques.

Un technique naturelle et très fréquemment utilisée pour remplir le résultat consiste à confondre chaque objet avec le vecteur de sa forme linéarisée (R2,R3). L'algorithme utilisé reste ainsi indépendant du rang des objets en présence (Alg.1) :

a,b,r,i sont des registres de travail, elcnt est le nombre d'éléments commun à A,B et R.

```
Début
    i:=0;
    tantque i < elcnt faire
        a:=charger(A,i);
        b:=charger(B,i);
        r:=additionner(a,b);
        ranger(r,R,i);
        incr i;
    fintant
fin
```

Algorithme 1 : l'addition (1 boucle)

Si l'on range les valeurs de chaque objet dans un **seul** bloc de la mémoire uniforme en respectant le même ordre de rangement, **l'ordre lexicographique** par exemple, les primitives "charger" et "ranger" s'explicitent très simplement. Appelons α, β et τ les adresses d'implantation des trois objets et λ le nombre de cellules nécessaires pour représenter un élément de A,B ou R :

a:=charger(A,i) $\leftrightarrow$ a reçoit le contenu de $(\alpha+\lambda x i)$

ranger(r,R,i) $\leftrightarrow$ r devient le contenu de $(\tau+\lambda x i)$

Nous pouvons utiliser ce **mode d'adressage** des éléments même si une optimisation de type partage existe entre A et B. Nous parlerons alors d' **adressage fonctionnel simple** .

I.1.2 - Réduction totale d'attributs

Dans le cas d'un opérateur scalaire, toutes les dimensions jouent le même rôle. La méthode précédente consiste à remplacer les attributs des opérandes par des attributs réduits équivalents :

Attributs de A,B et R	Attributs réduits de A,B et R
rang : p	rang : 1
rho : $r_0, r_1, \ldots, r_{p-1}$	rho : $R_0 = r_0 x w_0$ = elcnt
wv : $w_0, w_1, \ldots, w_{p-1}$	wv : $W_0 = 1$

Cette simplification, que nous appelerons réduction totale

d'attributs, s'applique à tous les opérateurs scalaires monadiques et dyadiques. On peut également s'en servir dans ,1,∈2,⍳2,⍴2,∘.f.

En revanche ce mode d'accès s'avère plus délicat d'emploi avec les autres opérateurs. Parmi ceux-ci, certains particularisent une des coordonnées (k) de l'opérande principal, appelée axe : ⌽1,⌽2,⌿1,⍀1,⌿2,⍀2,,2,⍒1,⍒2,⍋2,⍋2. Cette coordonnée est implicitement définie comme dans :

⊖B (k=0) ou ⍒3 2 4⍴A (k=2)

ou explicitement nommée comme dans :

+⌿[1]A (k=1) ou A,[2.5]B (k=2.5).

En partant d'un exemple, nous allons dégager une méthode générale permettant de produire des algorithmes simples et efficaces, ayant un nombre limité de boucles et n'utilisant que des additions pour la génération des indices du résultat et des opérandes dans de telles opérations.

I.1.3 - L'exemple de la compression :

Soit à évaluer l'expression, V/[K]A, dans laquelle :

V : 1 0 0 1

K : 1 (deuxième coordonnée)

r_a: 3 4 5

w_a: 20 5 1

La dimension du résultat le long de la coordonnée de compression vaut : (⍴R)[K]←+/V, ce qui donne pour le résultat :

r_r: 3 2 5

w_r: 10 5 1

La façon dont on obtient les éléments de R à partir de ceux de A est schématisée sur la fig.5. Pour l'ordre dans lequel on peut le faire, deux méthodes sont applicables :

Première méthode : balayer linéairement A en déterminant pour chacun de ses éléments à quelle valeur de V il correspond et, en fonction de cette valeur, remplir ou non le résultat.

Deuxième méthode : balayer linéairement le vecteur V et, pour chaque valeur vraie de V, sélectionner tout le sous-tableau de A correspondant pour en garnir le sous-tableau correspondant de R.

La première méthode permet d'accéder les trois objets avec un adressage fonctionnel simple, car A et R sont parcourus linéairement et V est connu pour être un vecteur. La deuxième ne le permet pas, car A et R sont accédés dans un ordre aléatoire a priori.

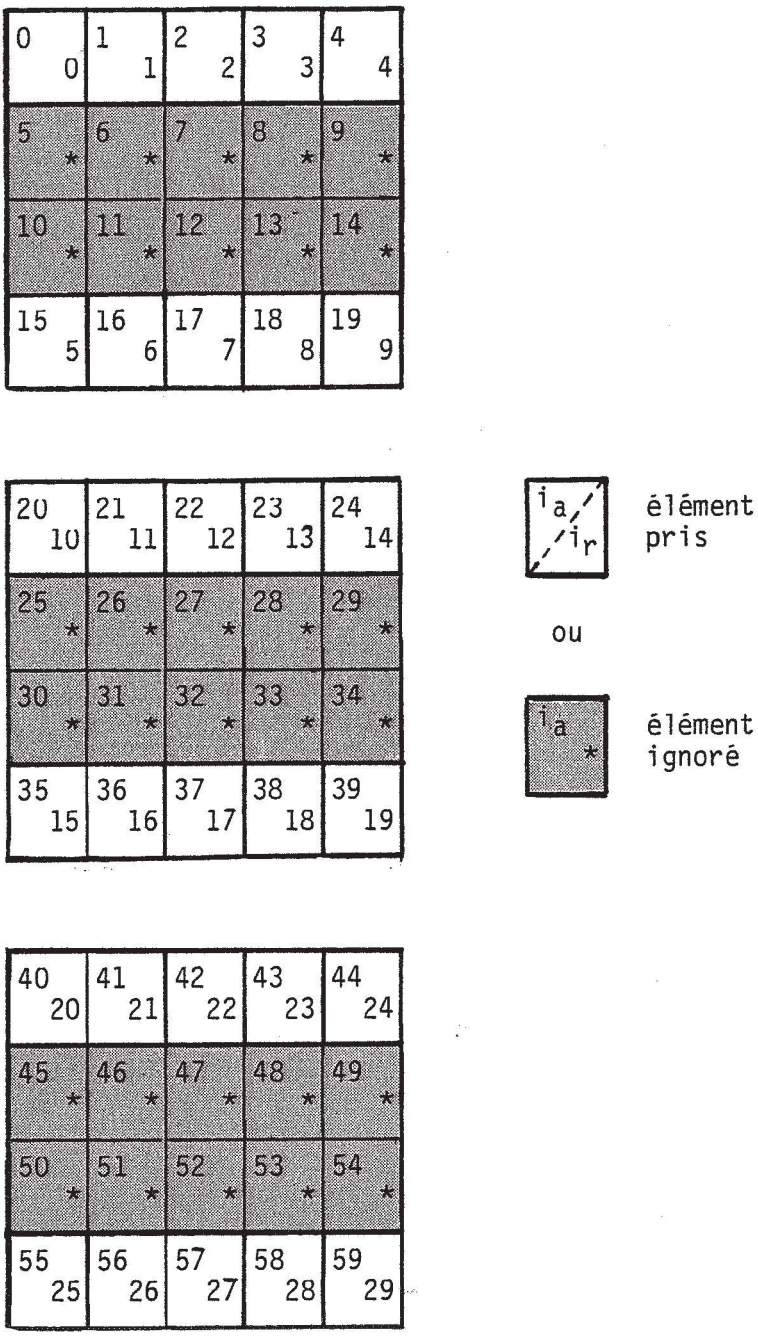


Figure 5 : *R*←1 0 0 1/[1]*A*←3 4 5⍴⍳60

Elle minimise cependant le nombre d'accès à V et à A. Nous présenterons donc l'algorithme de la deuxième méthode (Alg.2). On y utilise 9 indices :

$i_0 \in [0, r_0[$, $i_1 \in [0, r_1[$, $i_2 \in [0, r_2[$

la_0, la_1, la_2 qui intègrent en cascade les indices i_0, i_1, i_2 en déplacement par rapport à A, c'est à dire qui vérifient toujours :

($la_1 = i_1 . w_1$; $la_0 = i_0 . w_0 + i_1 . w_1$; $la_2 = i_0 . w_0 + i_1 . w_1 + i_2$)

lr_0, lr_1, lr_2 qui intègrent i_0, i_1, i_2 de la même façon sur R.

```
Début
   i1,la1,lr1:=0;
   tantque i1<r1 faire
      la0:=la1;lr0:=lr1;
      v:=charger(V,i1);
      si v alors
         i0:=0;
         tantque i0<r0 faire
            i2:=0;la2:=la0;lr2:=lr0;
            tantque i2<r2 faire
               ia:=la2;ir:=lr2;
               r,a:=charger(A,ia);
               ranger(r,R,ir);
               incr i2;incr la2;incr lr2;
            fintant
            la0:=la0+wa0;lr0:=lr0+wr0;
            incr i0;
         fintant
         lr1:=lr1+wr1;
      finsi
      la1:=la1+wa1;
      incr i1;
   fintant
fin
```

Algorithme 2 : la compression (3 boucles)

Nous constatons que nous pouvons rendre les boucles indépendantes de l'ordre du vecteur de dimensions, dans le cas précédent les indices varient dans l'ordre i_1, i_0 puis i_2. Dans les calculs d'indices ne figurent que des additions qui sont en général plus rapides à exécuter que des multiplications.

I.1.4 - La réduction partielle d'attributs

Cette méthode se généralise à des tableaux de rang supérieur ou inférieur par le calcul des attributs réduits par rapport à une certaine coordonnée (k).

Attributs de A :

rang : p

rho : $r_0, r_1, \ldots, r_{k-1}, r_k, r_{k+1}, \ldots, r_{p-1}$

wv : $w_0, w_1, \ldots, w_{k-1}, w_k, w_{k+1}, \ldots, 1$

Attributs réduits de A :

rang : 3

rho : $(r_0 x \ldots x r_{k-1}), r_k, (r_{k+1} x \ldots x r_{p-1})$ ou R_0, R_1, R_2

wv : $w_{k-1}, w_k, 1$ ou W_0, W_1, W_2

On évalue directement les attributs réduits d'un objet de vecteur de poids W, sachant que :

$\forall i \in [0, p[$, $w_i = r_{i+1} . w_{i+1}$ (par définition de W) et que

$elcnt = r_0 . w_0$ (nombre d'éléments)

on en déduit :

$W_2 = 1$, $W_1 = R_2 . W_2 = R_2 . 1 = w_k$, $W_0 = R_1 . W_1 = r_k . w_k$, $elcnt = R_0 . W_0$

$R_0 = elcnt / r_k . w_k$

d'où les attributs réduits de A en fonction de $(elcnt, r_k, w_k)$:

Attributs réduits de A :

rang : 3

rho : $elcnt / r_k . w_k, r_k, w_k$

wv : $r_k . w_k, w_k, 1$

Les mêmes calculs pouvant être appliqués aux attributs du résultat R, l'algorithme 1 devient applicable à des objets de rang supérieur ou inférieur.

Pour une troisième catégorie d'opérateurs, parmi lesquels figurent $\boxplus_1, \boxplus_2, \circledcirc_1, \circledcirc_2$ et [;;;], aucune réduction ne peut être opérée sur l'opérande principal.

I.1.5 - L'indexation et l'affectation indicée

Soit l'expression générale d'indexation :

$R \leftarrow A\,[IX_0;IX_1;\ldots;IX_{p-1}]$

Attributs de A :

rang : p

rho : $r_0, r_1, \ldots, r_{p-1}$

wv : $w_0, w_1, \ldots, w_{p-1}$

Les attributs du résultat seront d'après les règles de l'indexation :

Attributs de R :

rang : somme des rangs des IX_i

rho : concaténation des dimensions des IX_i

Si nous pouvons pratiquer des réductions totales d'attributs de tous les indices et du résultat, il nous est impossible de réduire ceux de A. Le nombre de boucles imbriquées de l'algorithme dépend du rang de A. On peut cependant dans chacune d'elles évaluer les quantités cumulées la_i afin de minimiser le nombre de multiplications (alg.3).

Remarques :

a) l'algorithme s'emploie également pour l'affectation indicée qui correspondrait à l'expression :

$A[IX_0;IX_1;\ldots;IX_{p-1}] \leftarrow R$

Seules les primitives "lire" et "écrire" différencient les deux opérateurs :

Primitives	Indexation	Affectation indicée
x:=lire;	x:=charger(A,i_a);	x:=charger(R,i_r);
écrire(x);	ranger(x,R,i_r);	ranger(x,A,i_a);

b) Le nombre de boucles étant inconnu, l'algorithme ne sera pas programmable sous cette forme directement.

```
Début
   i_r, i_0 :=0;
   tantque i_0<elcnt_0 faire
      I_0 :=charger(IX_0, i_0); la_0 :=I_0 x w_0; contrôler(I_0<r_0);
      i_1 :=0;
      tantque i_1<elcnt_1 faire
         I_1 :=charger(IX_1, i_1); la_1 :=la_0 +I_1 x w_1;
         contrôler(I_1<r_1);
         ...
         ...
            i_p-1 :=0;
            tantque i_p-1<elcnt_p-1 faire
               I_p-1 :=charger(IX_p-1, i_p-1);
               la_p-1 :=la_p-2 +I_p-1 x w_p-1;
               contrôler(I_p-1<r_p-1);
               i_a :=la_p-1;
               x:=lire; (* dans A ou dans R *)
               écrire(x); (* dans R ou dans A *)
               incr i_r; incr i_p-1;
            fintant
         ...
         ...
         incr i_1;
      fintant
      incr i_0;
   fintant
fin.
```

Algorithme 3 : l'indexation ou l'affectation indicée
(p boucles)

I.1.6 - Les 4 types de parcours

De l'étude précédente, nous concluons qu'au cours des opérations APL, les objets ne sont que très rarement accédés de façon totalement aléatoire. Il est possible en fait de dégager 4 types de parcours des opérandes.

1°) Parcours purement séquentiel : nous dirons aussi parcours linéaire et il sera employé dans les opérations scalaires, les produits externes, l'édition...

2°) Balayage le long d'une coordonnée (k) en majeure et parcours séquentiel de chaque sous-tableau sélectionné en mineure : ce type de parcours concerne par exemple les algorithmes de compression-expansion, de concaténation-laminage...

3°) Parcours séquentiel de tous les sous-tableaux déterminés par une coordonnée (k) en majeure et balayage le long de cette coordonnée en mineure : c'est le cas des algorithmes de réduction-balayage, du produit interne, de la rotation...

4°) Sélection aléatoire d'éléments de l'objet : il n'existe aucune relation simple entre l'élément accédé courant et l'élément à accéder suivant. Ce type de parcours nécessite de tenir à jour une liste d'indices et intervient dans les algorithmes de l'indexation-affectation indicée, de l'inversion de matrice...

Nous avons de plus la possibilité de réduire les attributs des objets non optimisés, totalement pour les algorithmes de type 1 et partiellement pour les algorithmes de type 2 ou de type 3; les objets optimisés restent, quant à eux, toujours irréductibles.

Dans la mise en oeuvre de tout opérateur, on s'efforcera, par le choix d'un algorithme approprié, d'une part de minimiser le nombre d'accès aux objets et d'autre part de balayer le plus d'objets ou de sous-objets de manière séquentielle. Le remplissage du résultat s'y prête en général fort bien.

I.2 - Influence des variables optimisées

Les algorithmes étudiés précédemment sont basés sur le principe que les éléments d'une variable sont rangés dans l'ordre lexicographique et de façon contiguë en mémoire. En d'autres termes, pour une telle variable, vecteur de poids et vecteur d'accès sont confondus, ce qui autorise les réductions d'attributs. Ces algo-

rithmes deviennent donc caducs sur des variables résultant d'un "beating".

Les valeurs d'une telle variable A s'obtiennent en accédant celles d'une autre variable A' rangée, elle, correctement. On utilise pour cela un triplet (p,abase,wv) :

p : rang de A

Abase : décalage en éléments à l'origine entre A et A'

W $(w_0,w_1,\ldots,w_{p-1})$: vecteur d'accès

Pour obtenir le i^e élément de A défini par le vecteur d'indices $I(i_0,i_1,\ldots,i_{p-1})$, on calcule :

i':= Abase + +/ W x I (1)

L'élément d'ordre i' de A' est l'élément désiré i de A.

Nous appellerons ce mode d'accès, **adressage fonctionnel calculé** : p multiplications et une addition s'intercalent par rapport à la fonction d'adressage simple précédente.

I.2.1 - Parcours séquentiel :

L'accès à la i^e valeur de A peut s'obtenir par divisions euclidiennes successives. Partant du vecteur de dimensions de A : $R(r_0,r_1,\ldots,r_{p-1})$, on évalue $I(i_0,i_1,\ldots,i_{p-1})$:

$$i = q_{p-1}.r_{p-1} + i_{p-1}$$
$$\ldots$$
$$\ldots$$
$$q_2 = q_1.r_1 + i_1$$
$$q_1 = q_0.r_0 + i_0$$

Ensuite, on applique la formule (1). On obtient finalement une fonction d'accès, $i_{(\text{de A})} \xrightarrow{\ f\ } i'_{(\text{de A}')}$, qui se formule en APL :

Abase + +/ W x R ⊤ i

a) - pile d'indices

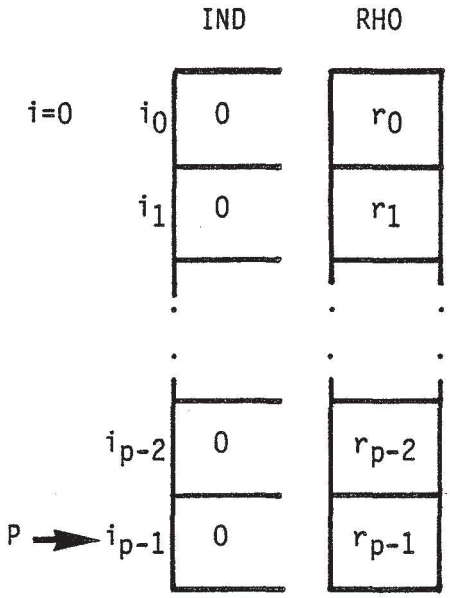
L'emploi d'une pile d'indices contrôlée par le vecteur des dimensions va permettre d'éviter toutes les divisions. Elle offre en plus l'avantage d'affranchir les algorithmes du rang des objets.

L'idée consiste à se munir d'une primitive "passage à l'élément suivant" : cette primitive agira sur une pile d'indices en

les faisant progresser dans l'ordre lexicographique.

b) - Initialisation d'une pile d'indices.

Les indices partent tous de 0. Un pointeur désigne l'indice variant le plus vite, celui que l'on fera toujours progresser en premier. Un indicateur "fin d'objet atteinte" permettra d'arrêter le parcours.




```
procédure Elton;
début
   j:=0;
   tantque j < p faire
      IND(j):=0;incr j;
   fintant
   OFF:=faux;P:=p-1;
fin
```

Pile d'indices et primitive d'initialisation

c) - Passage à l'élément suivant.

Chaque indice i_j varie dans le domaine $[0,r_j[$. La pile doit avoir un comportement cyclique, quand i_0 atteint la valeur r_0, ce qui provoque en plus la montée de l'événement "fin d'objet".

```
Procédure Elt;
début
    pt:=P; indic:=vrai;
    tantque indic faire
        incr IND(pt);
        si IND(pt)=RHO(pt) alors
            IND(pt):=0; decr pt;
            si pt<0 alors OFF:=vrai; indic:=faux; finsi
        sinon
            indic:=faux;
        finsi
    fintant
fin
```

I.2.2 - Parcours non séquentiel

a) - Aveuglement de la k^e coordonnée

Le but consiste cette fois pour une valeur donnée de l'indice i_k à balayer séquentiellement tous les éléments du sous-tableau ainsi déterminé. Nous utiliserons les deux primitives précédentes en "aveuglant" au préalable la k^e coordonnée. Ceci est obtenu en forçant la k^e dimension de A à 1 : RHO(k):=1. Ainsi l'appel de "Elt" provoquera uniquement une progression dans le sous-tableau i_k fixé. Cette méthode s'applique aux algorithmes de type 3 et 4.

b) - Accès aléatoire

Aucune primitive de parcours n'est nécessaire. Le remplissage direct des indices de la pile et l'utilisation de la fonction d'adressage (1) suffit.

I.2.3 - Optimisation et unification de la fonction d'adressage

La fonction d'adressage calculé présentée ne nous satisfait pas. Il est en effet illusoire d'économiser du temps de calcul - et de l'espace mémoire, certes - lors d' **une** opération de sélection fabriquant une variable optimisée, pour en reperdre dans **tous** les calculs qui y feront référence par la suite, chaque accès élémentaire exigeant le calcul de p multiplications.

L'étude des algorithmes intuitifs précédents sur des objets de rang fini et celle des travaux de D. Tusera (I18) nous a permis d'élaborer une nouvelle primitive de parcours, valable sur des objets de rang quelconque et qui permet d'éviter ces multiplications. Cette primitive unifie d'autre part les accès aux objets, qu'ils soient optimisés ou non.

La technique proposée s'apparente à l'adressage fonctionnel hiérarchisé employé par certains compilateurs ALGOL (G3) sous le nom de "dopé vector" ou "edge vector". La différence réside dans le calcul et non dans la mémorisation des vecteurs indicateurs des débuts de sous-tableaux.

Ecrivons tout d'abord l'algorithme virtuel du parcours séquentiel d'un objet quelconque de rang p. Cet algorithme réalise la fonction, f: $i \rightsquigarrow i'$, introduite en I.2.1 (Alg.4).

```
Début
   i,i0:=0;l0:=Abase;
   tantque i0<r0 faire
      i1:=0;l1:=l0;
      tantque i1<r1 faire

         ...

         ip-1:=0;lp-1:=lp-2;
         tantque ip-1<rp-1 faire
            i':=lp-1;
            écrire(i,'->',i');
            incr i;incr ip-1;lp-1:=lp-1+wp-1;
         fintant

         ...

         incr i1;l1:=l1+w1;
      fintant
      incr i0;l0:=l0+w0;
   fintant
fin
```

Algorithme 4 : Parcours séquentiel d'un objet (p boucles)

Cet algorithme est valable pour un objet non optimisé et réduit. Le cumul l_{p-1} le plus interne varie bien comme :

i' = Abase + +/ W x I

A la sortie de chaque boucle, le cumul de niveau supérieur devient la valeur de réinitialisation de tous les cumuls plus internes.

Nous avons déduit de cet algorithme les deux nouvelles primitives duales "Elt" et "Elton". Nous ajoutons pour cela, à la pile d'indices contrôlée par le vecteur de dimensions, une pile (CUM) associée au vecteur d'accès (WV) et réalisant la progression des cumuls.

a) - Initialisation, primitive "Elton" :

```
Procédure Elton;
Début
   j:=0;
   tantque j<p faire
      IND(j):=0; CUM(j):=Abase;
      incr j;
   fintant
   OFF:=faux; P:=p-1;
fin
```

Algorithme de la primitive "Elton".

b) - Passage au suivant : primitive "Elt"

Il devient légèrement plus compliqué, car il faut propager vers le bas les réinitialisations. Quand la fin d'objet est atteinte tous les cumuls ont repris leur valeur initiale "Abase".

```
Procédure Elt;
Début
   pt:=P; indic:=vrai;
   tantque indic faire
      incr IND(pt); c,CUM(pt):=CUM(pt)+WV(pt);
      si IND(pt)=RHO(pt) alors IND(pt):=0; decr pt;
         si pt<0 alors
            c:=Abase; OFF:=vrai; indic:=faux;
         finsi
      sinon
         indic:=faux;
      finsi
   fintant
   tantque pt<P faire
      incr pt; CUM(pt):=c;
   fintant
fin
```

Algorithme de la primitive "Elt" (Remarque : Seule une algorithmique structurée justifie la présence du booléen "indic").

Muni de ces deux primitives, on peut maintenant expliciter les algorithmes généraux de parcours.

c) - Parcours séquentiel d'un objet A quelconque :

```
Début
   Elton;
   tantque non OFF faire
      i_a:=CUM(p-1); a:=charger(A,i_a);
      Elt;
   fintant
fin
```

d) - Parcours de type 2 :

```
Début
   RHO(k):=1; Elton; i_k:=0; c_k:=0;
   tantque i_k<r_k faire
      tantque non OFF faire
         i_a:=CUM(p-1)+c_k; a:=charger(A,i_a);
         Elt;
      fintant
      OFF:=faux; incr i_k; c_k:=c_k+w_k;
   fintant
fin
```

e) - Parcours de type 3 :

```
Début
   RHO(k):=1; Elton;
   tantque non OFF faire
      i_k:=0; l:=CUM(p-1);
      tantque i_k<r_k faire
         a:=charger(A,l);
         l:=l+w_k; incr i_k;
      fintant
      Elt;
   fintant
fin
```

I.2.4 - Evaluation du gain

Nous comparons les deux méthodes d'accès, en nous limitant au parcours séquentiel. Soient a et m les temps nécessaires à l'exécution d'une addition et d'une multiplication.

Avec l'adressage fonctionnel calculé, on accède (x/ρA) éléments, ce qui demande en temps d'accès :

$T_{1a}=(p\ .r_0.r_1...r_{p-1})x(a+m)$

et pour la gestion de la pile d'indices :

$T_{1s}=(r_0+r_0.r_1+r_0.r_1.r_2+...+r_0.r_1...r_{p-1}).a$

L'utilisation de l'adressage fonctionnel simple et d'une pile de cumuls conduit aux temps :

$T_{2a}=0$ et $T_{2s}=2.T_{1s}$

Par exemple, le parcours séquentiel d'une matrice de dimensions 10x10x10 demandera, dans le premier cas :

4110 additions et 3000 multiplications

et dans le second cas :

2220 additions seulement.

I.3 - Influence de la mémoire virtuelle

Nous nous intéressons toujours à l'expression, R A+B, dans laquelle, cette fois-ci, A et B sont de très grosses variables.

I.3.1 - Le procès de la mémoire uniforme

Pour de nombreuses réalisations (R2,R3), la condition sine qua non d'acceptabilité d'une telle opération est la résidence complète dans la mémoire adressable du calculateur des blocs représentant les valeurs de A, B et R. D'autre part, ces réalisations pratiquent rarement la segmentation de données, ce qui entraîne qu'au moment de l'exécution, les valeurs de chaque objet sont représentées par un ensemble de cellules contiguës de la mémoire principale.

Remarquons que le nombre de variables simultanément accessibles peut être plus important. Le maximum est atteint avec l'indexation, $R \leftarrow A[I_0;I_1;...;I_{p-1}]$; il faut alors en mémoire accessible p index,le résultat et l'indexé (p étant la valeur limite du rang d'un objet imposée par la réalisation).

On voit d'entrée que cette méthode impose une limitation

sur la taille des objets, limitation qui, fâcheusement, varie selon l'opération (3 objets pour une addition, p+2 pour une indexation), selon les types de représentation utilisés, et même selon l'implantation des objets en mémoire si des techniques de retassage dynamique de la mémoire centrale du type ramasse-miettes n'ont pas été prévues lors de la conception de la gestion de mémoire.

Dans le cas de notre réalisation, la seule limitation provient de l'utilisation d'une arithmétique entière sur 16 bits. La taille d'un objet, comptée en nombre d'éléments, est donc limitée à 32767. La quantité d'octets nécessaire au stockage d'une telle variable complète dépend des types de représentation utilisés. Ainsi, dans le cas de l'addition, si A et B représentent des tableaux numériques de taille maximum, il faut pouvoir manipuler :

T = 3 x 32768 x 6 octets = 576 Ko = 0,5 Mo

Cette quantité dépasse largement la capacité de la mémoire adressable d'une petite machine et même, souvent, celle de la mémoire physique.

La résolution de ce problème d'accès passe par la mise en oeuvre d'une mémoire virtuelle simulée et du recouvrement complet des programmes et des données. Pour y parvenir, nous avons dû définir une gestion complète de l'espace adressable du processus interprète.

I.3.2 - La mémoire virtuelle paginée

Nous utilisons une mémoire virtuelle paginée. La taille des pages est directement imposée (cf chapitre 2) par le mécanisme de réimplantation dynamique de la mémoire adressable du matériel utilisé ; elle vaut 4 K-octets. Nous avons défini quatre catégories de mémoire virtuelle : programme, données actives, données fichiers et données de partage (G5). Chaque espace virtuel peut atteindre la taille de 4 millions d'octets.

I.3.3 - Le préchargement des pages

Le chargement à la demande en utilisant les défauts de page ne peut être mis en oeuvre. Nous utilisons donc la technique de **préchargement des pages** (G8,M5). Ceci signifie que l'interprète gère complètement ses accès afin de toujours connaître les identifications des pages qui sont en **configuration** dans l'espace adressable de la machine.

a) - Recouvrement des programmes : Le préchargement des pages programme est mis en oeuvre directement dans les programmes de

l'interprete par une technique très classique de recouvrement, les différentes parties fonctionnelles de l'interprète étant organisées sous la forme d'une arborescence, une branche étant contenue dans une page.

b) - Recouvrement des données : La technique consiste, à première vue, pour accéder une certaine partie d'une donnée, à déterminer qu'elle est la page intéressée par l'accès, à calculer le déplacement relatif dans cette page et enfin à trouver dans quelle case de la mémoire logique est implantée cette page (éventuellement, il faudra implanter la page dans une case non occupée, voire chasser une autre page de la configuration). L'addition de l'adresse logique d'implantation et du déplacement relatif précédent fournit finalement une adresse absolue valable au moins pour un accès.

Cette méthode apporte une solution à notre problème : nous pourrons appliquer une opération sur un "gros" opérande, les pages des derniers éléments chassant celles des premiers accédés. Nous pourrons également traiter une indexation avec de nombreux indices, même si les valeurs de chaque indice sont situées dans des pages toutes différentes.

Cependant, elle s'oppose à nos soucis de performances. En effet, ces calculs et recherches coûteux en temps pénalisent toutes les opérations sur des "petits" objets (qui peuvent d'ailleurs appartenir à la même page). Nous voulons pouvoir additionner deux vecteurs de 192 K-octets chacun, mais la multiplication de deux matrices de 2000 entiers doit rester aussi performante que celle qui serait réalisée dans une mémoire uniforme.

I.3.4 - Le structurateur de page

Nous distinguons deux sortes d'accès, associés dans notre esprit à deux catégories d'objets :

a) - Accès par verrouillage d'une page dans l'espace adressable : la page restera en configuration, là où elle a été implantée, jusqu'à son déverrouillage complet. L'adresse absolue des objets d'une page verrouillée est donc valide jusqu'à nouvel ordre. Les accès par verrouillage correspondent aux petits objets (dont la taille est inférieure à 4K-octets).

b) - Accès temporaires à une page : la page est rendue physiquement présente en mémoire, mais peut disparaître après quelques autres accès temporaires ou demandes de verrouillage. Ces accès cor-

respondent aux gros objets (dont la taille dépasse 4K-octets).

c) - Le structurateur de page : un processus noyau est chargé de la gestion complète des cases de l'espace adressable, nous l'appelons le structurateur de page.

Il distingue trois catégories de cases dans l'espace adressable : les cases libres, les cases verrouillées et les cases temporaires, ensemble qu'il gère de la façon suivante :

Cases libres : toutes les cases sont initialement libres et allouées en priorité.

Cases verrouillées : elles correspondent à des pages verrouillées et le gestionnaire leur associe un compteur de verrouillage (plusieurs petits objets peuvent être contenus dans la même page) qu'il incrémente au verrouillage et décrémente au déverrouillage : il sait ainsi quand une page n'est plus du tout bloquée.

Cases temporaires : le structurateur s'efforce de conserver en configuration les pages qui ont le plus de chance d'être accédées dans le futur immédiat. Dans l'ignorance de la chaîne des références à venir - c'est à dire de la succession des prochains accès aux pages virtuelles (M5,G8) - il se base sur l'utilisation passée des pages en mettant en oeuvre un algorithme de remplacement du type L.R.U. ("Least Recently Used" Algorithm). A tout accès temporaire à une page donnée, celle-ci est déclarée la plus récemment utilisée. Afin de ne pas perdre des informations de l'espace de travail, une page complètement déverrouillée entrera dans cette file L.R.U..

Quand une page demandée pour un accès temporaire ou verrouillant sera absente de la configuration, nous choisirons de chasser de l'espace de travail la page correspondant à la case la moins récemment utilisée.

II - LES LIENS

L'étude précédente nous montre que les accès aux objets peuvent revêtir des formes diverses en étant influencés par :

1) la façon dont on parcourt les objets, les requérants d'accès, souvent des opérateurs, optant pour l'accès séquentiel ou aléatoire

2) la multiplicité des représentations de variables - scalaires, j-vecteurs - et la nécessité de conversions dynamiques imposées par l'existence de plusieurs types numériques

3) la taille des objets et la disponibilité de la ressource mémoire adressable.

Afin d'affranchir l'interprétation de cette diversité, nous pouvons concevoir une seule fonction d'accès, très générale, qui tiendra compte de ces paramètres : l'accès à l'élément i d'un objet sera en quelque sorte "interprété". Une telle fonction est complexe à réaliser et peu performante : les questions 1, 2 et 3 se reposant à chaque accès sous la forme de tests d'exécution. Nous avons préféré compiler dynamiquement ces accès.

II.1 - La compilation des accès

Cette solution consiste à se poser les questions 1, 2 et 3 une fois pour toutes avant tout travail requérant suffisamment d'accès et à générer dynamiquement une procédure d'accès. Une telle procédure intègre dans son code tous les problèmes précédemment étudiés. Une séquence d'accès se décompose alors en deux phases :

- la génération d'un **lien** à un objet ou production d'un sous-programme d'accès spécifique,
- les accès proprement dits à cet objet ou appel du lien généré.

II.2 - Justification

Compiler un sous-programme d'accès étant une opération relativement importante, on peut se demander quel est le gain de temps réalisé ? En fait, le gain est d'autant plus intéressant que le code sera appelé plus souvent.

Ainsi, pendant toute l'analyse syntaxique d'une ligne APL source et la génération du code interne correspondant à celle-ci, on utilisera deux liens sur la table des symboles. De même, durant toute une session, un lien permet d'accéder la table des descripteurs.

En fait, pour tout opérateur APL, on peut dissocier le temps nécessaire à son exécution en deux phases bien précises (I10) :

- la phase de mise en place : contrôles de compatibilité des opérandes, évaluation des attributs du résultat, demande de mémoire, préparation des accès, initialisation du travail (zones temporaires, tables etc.). Le temps nécessaire à cette première phase est pratiquement fixe pour un opérateur donné : Δ .

- la phase d'itération : calcul des valeurs du résultat terme à terme. Le temps pris par cette phase dépend alors de l'opérateur, de l'algorithme utilisé et surtout des nombres respectifs d'éléments du ou des opérandes en présence. Appelons δ , le temps d'accès à un élément, n_1, n_2 et n, les nombres d'éléments de l'opérande gauche, de l'opérande droit et du résultat. On obtient :

1˚) pour l'addition ($n=n_1=n_2$) : $t=\Delta+3.n.\delta$

2˚) pour un tri arborescent (en nLogn) : les valeurs sont rangées dans une zone temporaire, puis relues, ce qui donne ($n=n_2$) :

lecture de l'opérande, n accès,

écriture du résultat, n accès,

descente, remontée des nombres dans l'arbre, 2nLogn accès.

Au total, $t=\Delta+2.(n+1).Logn.\delta$

3˚) pour certains opérateurs le nombre d'accès dépendra des valeurs rencontrées dans l'objet (ϵ_2, ι_2).

Hormis ces derniers cas, pour lesquels on peut néanmoins déterminer une borne supérieure, la règle générale s'énonce sous la forme :

$t=\Delta+\delta.T(n_1,n_2,n)$

où T mesure l'ordre de l'algorithme utilisé.

En compilant les accès, nous perdons du temps dans la première phase par une augmentation de la constante additive Δ , mais nous espérons en regagner beaucoup plus lors de la seconde phase en diminuant fortement le facteur constant δ . D'un point de vue pratique, ce gain n'est pas aisé à évaluer. Pour une application donnée, le gain dépend étroitement de la nature du travail à effectuer - traitement de grosses données par exemple - et du style de programma-

tion de l'utilisateur.

Cette étude d'accès nous permet de dégager les paramètres nécessaires à la génération d'un lien et en fonction de ceux-ci de ranger les liens par classe.

II.3 - Classification des liens

II.3.1 - Liens sur structures de données de l'interprète

De nombreuses structures de données nécessaires à l'interprétation sont réalisées à l'aide de mémoire APL (table des symboles, des descripteurs, pile des contextes, etc.). Seuls les gérants de telles structures, qui connaissent leur organisation interne, peuvent y avoir accès. L'intérêt de générer des liens sur ces structures réside dans le fait que leurs gérants deviennent indépendants de leur implantation dans la mémoire virtuelle. L'augmentation de la taille de la table des symboles en zone active, par exemple, n'est plus assujettie à une série de manoeuvres compliquées comme sous certains systèmes (R1,R2,R3,R4).

II.3.2 - Liens sur variables

En général, l'accès aux variables se fait sous le contrôle de la machine d'exécution qui rassemble alors l'ensemble de la description de la variable dans un descripteur statique (résident) unique : le **descripteur éclaté** . En fonction des informations rencontrées dans ce dernier, des liens très divers sont générés :

1°) Variables vides : une structure APL vide n'a pas de valeur. La demande d'une procédure d'accès sur une telle variable est toujours acceptée pour des raisons de régularité des algorithmes. Ce que doit produire le lien généré est plus sujet à débat. Parmi les diverses solutions possibles (erreur de valeur, anomalie système, élément de remplissage, etc.), nous avons choisi a priori de compiler un sous-programme ineffectif (un retour).

2°) Scalaires et pseudo-scalaires en lecture seule : les procédures compilées sont des fonctions constantes des indices. Ces liens accèdent directement le descripteur éclaté. Dans le cas des pseudo-scalaires, un seul accès est opéré à la génération du lien,

puis le descripteur éclaté est transformé en un descripteur de scalaire.

3°) J-vecteurs (en consultation uniquement) : en cas d'accès séquentiel, l'emploi de la fonction d'adressage par cumul fournit directement la i^e valeur du j-vecteur, $\underline{j}$(elcnt,base,pas) :

CUM(0)=1= base+i_0.pas

La procédure d'accès à $\underline{j}$ est donc l'identité.

a:=charger($\underline{j}$,1) $\leftrightarrow$ a:=1

En cas d'accès aléatoire, la paramètre passé au lien est l'indice d'accès, i_0 :

a:=charger($\underline{j}$,i_0) $\leftrightarrow$ a:=base+i_0.pas

4°) Variables à valeurs en mémoire : l'accès aux éléments passe par celui de la mémoire adressable. Le générateur compilera en fonction de la taille de l'objet et de l'actuelle utilisation de la mémoire un lien **rigide** ou un lien **souple** .

a) Lien rigide : à la création de ce lien, la seule page où sont rassemblées toutes les valeurs de la variable est verrouillée dans la configuration de l'interprète. L'adresse d'implantation de la variable est compilée directement dans ce lien. Quel que soit l'élément accédé, le lien n'a pas à vérifier la disparition de la page. Ces liens rigides sont donc aussi performants que des routines d'accès dans une mémoire uniforme.

b) Lien souple : un tel lien doit s'assurer à chaque accès - pour tout i - de la présence de la page concernée - f(i) - dans la configuration, chaque accès aléatoire pouvant a priori solliciter une page distincte. Ces liens sont plus complexes et moins performants, ce qui n'est jamais qu'une conséquence de la facilité offerte de manipulation de grosses données.

5°) Classes d'accès logiques : nous distinguerons deux genres d'accès logiques.

a) Accès logique séquentiel (ou linéaire) : l'ensemble des valeurs constituant la donnée est considéré comme un vecteur. Un seul indice i_1 suffit à désigner un élément (adressage fonctionnel simple). Un lien de ce type servira aussi à accéder aléatoirement un vecteur. Cette classe de lien est la seule envisageable pour les accès aux structures de données.

b) accès logique aléatoire : Le requérant d'accès veut pouvoir accéder les éléments d'une structure à plusieurs indices, et dans un ordre aléatoire. La liste d'indices de l'élément désiré

$(i_0, i_1, \ldots, i_{p-1})$ proposée au lien lui permettra de calculer un indice séquentiel i_1 en appliquant la formule de l'adressage fonctionnel calculé :

$$i_1 := Abase + \sum_{j=0,p-1} w_j . i_j$$

6°) Type de représentation des variables : au sein d'une même variable, tous les éléments sont représentés de la même façon (homogénéïté de type). Appelons to ce type. Pour localiser l'élément d'indice logique i_1 d'une variable de type to, implantée à l'adresse a_0, le lien calculera l'adresse de l'élément :

$a := f(a_0, i_1, to)$

7°) Fonctions d'accès : à la génération d'un lien, un mode précisera la fonction d'accès demandée.

a) Lien de lecture : un tel lien retournera une valeur. Tous les mélanges de types numériques étant possibles, le requérant d'accès devra préciser quel est le **type à fournir** , les liens se chargeront ainsi automatiquement de toutes les conversions implicites. Des expressions sémantiquement valides, comme 1='A', nous obligent à pouvoir créer un lien numérique sur un objet de type caractère.

b) Lien en écriture : on utilisera ces liens pour créer de nouveaux objets et pour modifier sélectivement certaines valeurs d'un objet déjà existant. Nous n'avons pas choisi de pouvoir convertir les valeurs avant de les écrire. Par exemple, un opérateur évalue toujours le type du résultat t_r, en fonction des types t_1 et t_2 des deux opérandes en présence, quand il ne l'impose pas :

$t_r := fonc(opérateur, t_1, t_2)$

Les trois liens requis par l'opérateur seront caractérisés par :

	mode	type	conversion éventuelle
opérande 1 :	lecture	t_r	$t_1 \longrightarrow t_r$
opérande 2 :	lecture	t_r	$t_2 \longrightarrow t_r$
résultat :	écriture	t_r	(néant)

II.4 - Le générateur de liens

Le générateur de liens est un processus noyau de l'interprète. Deux primitives permettent son activation : elles correspondent à une demande d'accès sur un objet et à la libération de cet accès.

II.4.1 - Demande d'accès, la primitive "RQST" :

L'accès peut être demandé sur un objet APL quelconque. Nous ne présentons que la syntaxe de cette primitive pour une demande d'accès sur une variable (cas plus général) :

proc:=RQST(de,td,fonc,indice,accès);

avec:

- de : descripteur éclaté de la variable
- td : type désiré (en lecture)
- fonc : fonction d'accès (lecture ou écriture)
- indice: indice i_1 ou pile indices $(i_0,i_1,\ldots,i_{p-1})$
- accès : classe d'accès (séquentiel ou aléatoire)

Cette primitive rend une procédure d'accès (proc) directement utilisable.

II.4.2 - Libération d'accès, la primitive "RLSE" :

Cette fonction permet de détruire un lien établi par les soins de la primitive précédente :

RLSE(proc);

II.4.3 - La mémoire de programmation des liens :

Le compilateur de liens génère du code directement exécutable dans une mémoire de programmation située en zone de donnée résidente. Plusieurs liens pouvant être actifs simultanément, il se pose le problème de l'allocation de portions de cette mémoire.

Partant de l'idée intuitive qu'un lien restait un sous-programme très simple, donc très court, nous avons procédé à une simulation en APL du générateur. Cette façon de faire présente deux intérêts :

- Elle a permis de valider l'architecture du compilateur qui a été réécrit très rapidement sur la machine hôte.
- La génération de toutes les combinaisons possibles d'accès a démontré empiriquement que la taille d'un lien n'excède pas 15 mots.

Assurés du fait que la mémoire de programmation peut être allouée par blocs d'une taille fixe, nous avons géré celle-ci sous la forme d'une liste d'entrées de 16 mots consécutifs, les entrées occupées contiennent des sous-programmes d'accès actifs, les entrées libres sont chainées entre elles.

II.4.4 - La compilation d'un lien :

Cette compilation revêt un caractère un peu particulier, dans la mesure où elle n'est pas sous le contrôle d'un analyseur syntaxique travaillant sur un texte source, mais dépend **directement** d'un nombre fini de paramètres.

Ces paramètres sont passés au générateur, ils décrivent alors l'utilisation que l'on désire faire de l'objet (type désiré, mode, accès, indice); les autres sont déterminés par le générateur lui-même en fonction des informations rencontrées dans la description de la variable et des paramètres précédents :

- to : le type de l'objet
- aconv:=f(fonc,to,td) : la procédure de conversion

Le paramètre "accès" peut prendre 3 nouvelles valeurs, ce qui donne au total 5 ossatures de lien possibles : vide, scalaire, j-vecteur, séquentiel ou aléatoire.

Dans les deux derniers cas, on détermine le paramètre mémoire m (rigide ou souple), en fonction de la taille de l'objet et des disponibilités de la mémoire adressable.

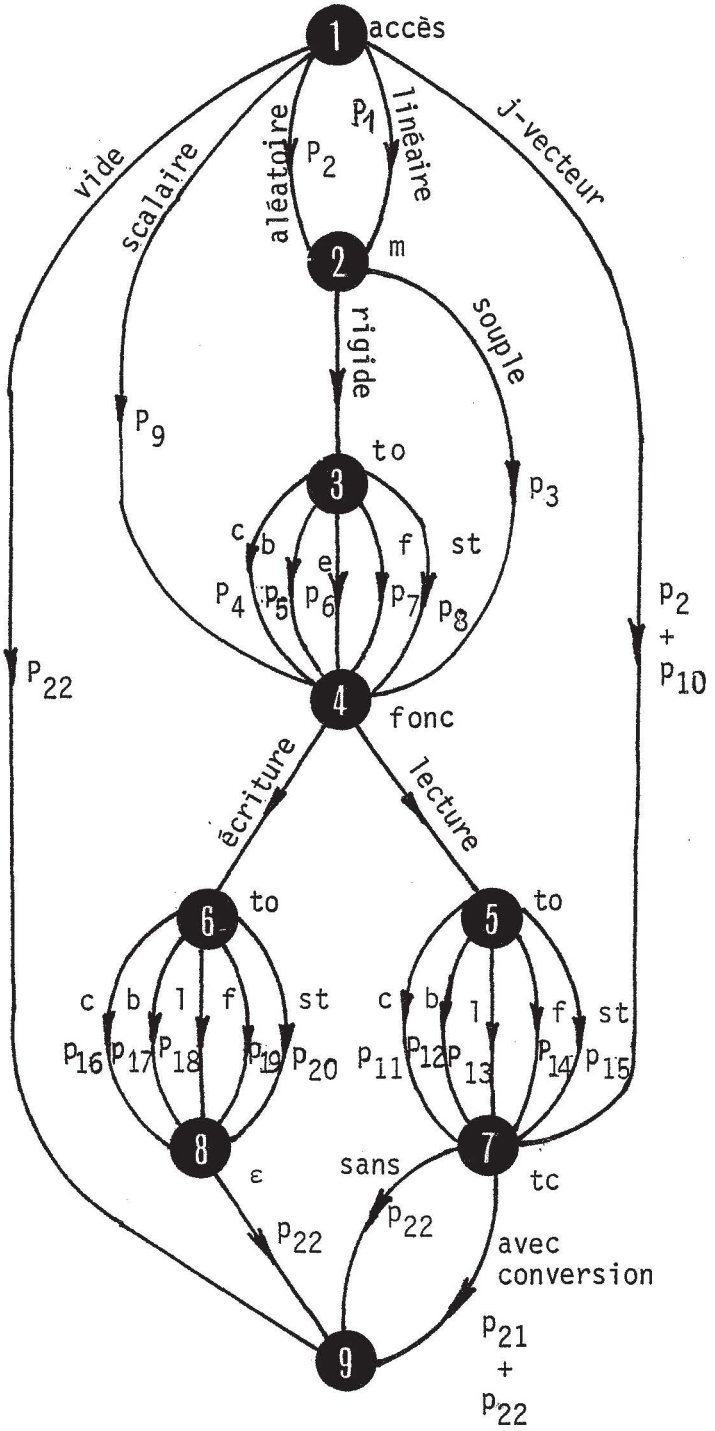
La première version du compilateur utilisait un petit automate permettant de générer un lien en fonction des paramètres précédents. Si cette méthode avait l'avantage d'une programmation élégante, elle n'était pas en revanche très performante, car elle suppose a priori toute suite d'instructions possibles générable alors qu'en fait la génération d'un lien précis (sur un j-vecteur, par exemple) suit un squelette de génération connu. En rendant ce compilateur moins général, c'est à dire plus conscient des objets qu'il traite, nous gagnons en performance. Cette spécificité a été obtenue en intégrant au niveau même du code du générateur les choix de génération possibles sous la forme de tests directs d'exécution.

Dans l'automate de génération présenté à la figure 6, nous employons 4 registres de travail (que l'on s'efforcera de confondre dans leur topographie de réalisation avec ceux de la machine hôte) :

i_l : indice logique d'élément
i_{phi} : pointeur physique sur l'élément
i_b : index supplémentaire pour les booléens
val : accumulateur général

et plusieurs références d'adresses :

indice : adresse de l'indice ou de la pile d'indices
abase,wv,adata,scal : adresses d'information du descripteur



Calcul de i_1

Calcul de i_ϕ et i_b

Lecture ou Ecriture de val

Conversion Retour

Paramètres : accès : j-vecteur, linéaire, non linéaire, scalaire, vide
m : rigide, souple
to : caractère, booléen, entier, réel, structure
fonc : lecture, écriture
tc : avec, sans conversion
ε : (transition vide)

Figure 6 : Automate de génération d'un lien.

Le code généré à chaque transition dans l'automate est rédigé dans un langage symbolique que nous supposons suffisamment clair pour ne pas avoir à le décrire. Ainsi :

==> $\{i_{phy} := \alpha_0 + 8 \times i_1\}$ signifie : "générer le code affectant au registre i_{phy} le résultat de l'opération..." et (x) représente l'information stockée à l'adresse x.

Les procédures sémantiques utilisées sont décrites dans la liste ci-dessous :

P1 : ==> $\{i_1 := (indice)\}$

P2 : ==> $\{i_1 := (abase) + \sum_j (wv[j]) \times (indice[j])\}$

P3 : ==> $\{i_{phy}, i_b := ACCES((adata), to, i_1)\}$

P4 : ==> $\{i_{phy} := \alpha_0 + i_1\}$

P5 : ==> $\{i_{phy} := \alpha_0 ;\ i_b := i_1\}$

P6 : ==> $\{i_{phy} := \alpha_0 + 2 \times i_1\}$

P7 : ==> $\{i_{phy} := \alpha_0 + 6 \times i_1\}$

P8 : ==> $\{i_{phy} := \alpha_0 + 8 \times i_1\}$

P9 : ==> $\{i_{phy} := scal\}$ to:=max(to,entier)

P10 : ==> $\{val := i_1\}$

P11 : ==> $\{val := charger\text{-}caractère(i_{phy})\}$

P12 : ==> $\{val := charger\text{-}booléen(i_{phy}, i_b)\}$

P13 : ==> $\{val := charger\text{-}entier(i_{phy})\}$

P14 : ==> $\{val := charger\text{-}réel(i_{phy})\}$

P15 : ==> $\{val := charger\text{-}structure(i_{phy})\}$

P16 : ==> $\{ranger\text{-}caractère(val, i_{phy})\}$

P17 : ==> $\{ranger\text{-}booléen(val, i_{phy}, i_b)\}$

P18 : ==> $\{ranger\text{-}entier(val, i_{phy})\}$

P19 : ==> $\{ranger\text{-}réel(val, i_{phy})\}$

P20 : ==> $\{ranger\text{-}structure(val, i_{phy})\}$

P21 : ==> $\{val := aconv(val)\}$

P22 : ==> {retour}

Procédures sémantiques utilisées.

PARTIE III

les EXTENSIONS

Cette troisième partie présente deux extensions originales du système APL/857.

Le chapitre cinquième recense et classe les diverses extensions proposées pour APL en extension "langage" et extension "système". Parmi celles réalisées, une extension de chaque catégorie est exposée et fait l'objet des deux chapitres suivants.

Le sixième chapitre justifie l'introduction des tableaux généralisés, présente parmi les nombreuses approches existantes celle qui a été choisie et expose les problèmes qui ont été rencontrés dans l'intégration de ces nouvelles structures au système.

Le dernier chapitre concerne la sûreté et la protection des applications. Parmi les recherches poursuivies dans cette voie, sont retenus les efforts faits en matière d'invisibilité des noms, de contrôle d'accès aux variables, de dépose de sceau sur les zones de travail et plus particulièrement d'outils de contrôle et de recouvrement des erreurs et interruptions. Dans chacun de ces cas, l'option introduite dans le système est exposée et critiquée.

CHAPITRE 5

INTRODUCTION

D'une manière générale, une extension vise à fournir à l'utilisateur un service jusqu'alors inexistant, ou à lui permettre d'accomplir plus facilement un traitement jusqu'alors compliqué ou inefficace.

La qualité d'extensibilité d'un système APL n'est pas une vertu en soi, le meilleur moyen de construire un système extensible est encore de prévoir la nature de ses extensions.

Sans entrer dans les détails, nous ferons un bref survol des extensions déjà existantes ou proposées, en nous limitant aux plus sérieuses. Nous laisserons de côté les inévitables gadgets hérités de chaque spécialiste de tel domaine informatique.

Deux extensions particulières ont retenu notre attention et sont étudiées plus en détail.

I - EXTENSION DU LANGAGE

I.1 - Structures de contrôle

Le but de ces nombreuses propositions est de munir le langage de structures de contrôle du type ALGOL ou PL1 (if-then-else, case of, do, while etc.). Citons APLGOL (I17) et EXEL-APL (U4).

I.2 - Nombres complexes

Le but est de rajouter un type numérique explicite complexe. Toutes les primitives ne sont pas facilement extensibles sur ce nouveau domaine (T1). Le problème des conversions explicites resurgit sous un jour nouveau : par exemple, l'expression, ¯1*.5 doit-elle fournir un nombre complexe ?

I.3 - Introduction de tableaux généralisés

Ce domaine a été très fortement exploré et présente beaucoup d'intérêt. Il fait l'objet du chapitre 6.

I.4 - Fonctions et variables système

Introduites par IBM dans APL-SV (L8), les fonctions système fournissent à l'utilisateur un moyen de contrôle et d'action sur son environnement. Les noms de ces objets commencent par un "quad" (⎕),

pour éviter les conflits de nom avec les objets de l'utilisateur.

L'adaptation de l'exécutant à l'activation de ces fonctions système et la fourniture de moyens logiciels permettant d'en créer de nouvelles sont des gages d'extensibilité. En effet, la plupart des extensions système sont réalisées par le biais de ces fonctions.

II - EXTENSIONS SYSTEME

Ces extensions visent à améliorer le système APL ou à l'ouvrir vers l'extérieur.

II.1 - Temps réel et traitement par lot

L'apport dans le système APL de primitives de création, destruction, activation et synchronisation de tâches, sous la forme de fonctions système, ouvre la porte aux applications temps réel. Le système automatique de contrôle d'exécution (A.C.E.) d'APL*PLUS est un modèle du genre (P1). Signalons toutefois qu'une telle extension est étroitement conditionnée par les possibilités logicielles et matérielles du système hôte utilisé.

II.2 - Extensions graphiques

On trouve dans la littérature deux types d'approches aux problèmes graphiques.

L'écriture en APL de l'ensemble des programmes de gestion, d'entrée ou de sortie d'images offre l'intérêt de fournir des applications graphiques portables (D1). En revanche, elles sont relativement inefficaces.

La seconde approche, plus intéressante, consiste à introduire dans l'interprète lui-même des opérateurs spécialisés. Une partie du travail est effectuée à l'intérieur du système et les temps deviennent alors acceptables. Une fois de plus la fonction système sera utilisée. A titre indicatif, nous renverrons le lecteur vers deux réalisations intéressantes (D2,D3).

II.3 - Outils logiciels de création de primitives, opérateurs ou fonctions système

Fournir à l'utilisateur de tels outils est à notre avis la meilleure façon de lui permettre de résoudre ses problèmes particuliers. Deux approches méritent l'attention :

a) implémentation de nouvelles primitives ou de nouveaux opérateurs en APL (code mou) : l'utilisateur développe dans le langage une fonction définie répondant précisemment à ses besoins. Puis il en fait une primitive du langage en utilisant un outil spécial d'intégration.

b) fonctions machine (code dur) : les fonctions machine sont des objets contenant essentiellement des instructions machine qui peuvent être exécutées sous le contrôle de l'interprète pour le compte de l'utilisateur possédant ces objets (R2,M4).

Quand une telle fonction suscite l'intérêt général, son intégration au système consiste à la transformer en une fonction système en lui attribuant un nom (□XX) et en connectant ce nom au code de la fonction. Celle-ci devient utilisable par tous.

Dans les deux cas (a) et (b) l'utilisateur crée lui-même ses propres fonctions. La caractéristique commune de celles-ci est qu'elles se comportent en tous points comme des fonctions définies verrouillées : elles peuvent avoir zéro, un ou deux paramètres d'appel, rendre ou non un résultat, être copiées, sauvegardées, détruites mais ni interrompues, ni listées.

II.4 - Sûreté et protection des applications

Sous ce titre, nous regrouperons tous les efforts tendant à fournir des outils permettant de développer en APL des applications **fermées** et **sûres** . Ceci fera l'objet du chapitre 7.

II.5 - Accès à des données extérieures à la zone de travail

Ce terme générique "d'accès à des données extérieures à la zone de travail active" recouvre tous les développements qui ont été faits pour améliorer l'accès de l'utilisateur à **son environnement** . La multitude des réalisations traduit, en l'absence de standard, le besoin éprouvé.

- Accès à des fichiers de données : les approches consistent à permettre l'accès depuis APL à des fichiers classiques séquentiels, séquentiels-indexés ou aléatoires, ou à des fichiers APL spécialisés à accès relatif.

Dans le premier cas, une interface avec le système de gestion de fichier standard est introduite dans APL, ce qui autorise la communication de données entre APL et les autres processeurs du système (M4).

Dans le second cas, un système orienté APL dans sa philosophie est créé de toutes pièces : un fichier étant considéré comme une **liste de composantes** , chaque composante représentant une variable APL **quelconque** . Suivant la sophistication des méthodes d'accès, on peut lire, écrire, ajouter, détruire ou modifier des composantes (F1,F2,F3) ou même effectuer des compressions, expansions ou rotations sur ces listes (U5). Là encore, on emploie le plus souvent les fonctions système. Les systèmes de fichiers du type APL*PLUS semblent s'être imposés dans ce genre aux yeux de nombreux utilisateurs.

- Accès à des variables partagées : une classe importante des fonctions système concerne le partage de variables entre deux processus, que ces processus représentent des utilisateurs APL ou des processus extérieurs à APL, comme TSIO (V1) qui permet l'accès, ainsi, à tous les types de fichiers gérés par le système d'exploitation.

C H A P I T R E 6

I - INTRODUCTION

Dans le courant de la dernière décennie, la communauté APL s'est beaucoup intéressée à l'introduction dans le langage de nouvelles structures de données : les tableaux généralisés.

L'objectif est de pouvoir créer des structures plus complexes et plus générales que les tableaux rectangulaires, denses et homogènes en type que nous connaissons.

Si les propositions d'extension dans ce domaine paraissent relever d'une véritable bataille d'experts (T2,T3,T4,T5,T6,T7,T8,T9 ,T12), bataille dans laquelle nous n'avons nullement la prétention d'intervenir, il est cependant réconfortant de constater que, dans l'ensemble, toutes reflètent une même tendance qui consiste à :

- définir des données récursives et hétérogènes : les éléments de ces données pourront être des tableaux de type différent
- munir le langage de nouvelles primitives et de nouveaux opérateurs
- étendre, voire redéfinir plus généralement le champ d'action des primitives existantes (extension "propre").

II - INTERETS DES TABLEAUX GENERALISES

C'est la première question que l'on peut se poser. Cette extension permet la manipulation de données logiquement liées entre elles, mais non représentables sous la forme de tableaux complets, car elles ont des attributs - rang, type ou dimensions - incompatibles. Remarquons que ce besoin a donné naissance à des organisations de données très diverses, citons pour mémoire :

- les groupes d'APL/360 (L8)
- les packages de SHARP APL (L10)
- les arbres d'APL-LAVAL.

Ces organisations demeurent cependant des organisations de **noms** .

Une telle innovation apporte à l'utilisateur la possibilité de créer et de manipuler des **données de structure arborescente** . On retrouve en un sens les articles de COBOL ou de PASCAL. Lorsqu'elle s'insère en plus dans une réalisation comportant une mémoire virtuelle, elle ouvre la porte aux fichiers et plus généralement aux bases

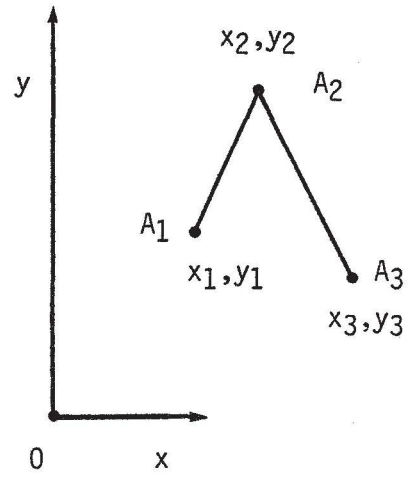
de données (G8). De nombreux exemples d'applications sont décrits dans la littérature :

- simulation d'un système de fichiers à accès direct organisés en liste de composantes (T5)
- réalisation du système de base de données relationnelle G.E.S.O.P. à l'aide de tableaux généralisés (T10).

Les tableaux généralisés trouvent encore des utilisations dans les applications graphiques ou en analyse des langages (T11).

II.1 - Représentation de dessins

Un dessin se représente couramment comme une ligne brisée définie par la donnée numérique des coordonnées des points qui la composent. Typiquement en APL, ce sera une matrice numérique nx2 :



x_1	y_1
x_2	y_2
x_3	y_3

Dessin Matrice APL

Une image complexe est plus facile à exploiter lorsqu'on la considère comme une composition de sous-images indépendantes plus simples. Les opérateurs graphiques d'APLIXI utilisent pour cela les tableaux généralisés.

II.2 - Utilisation de fonctions machine étrangères

Les tableaux généralisés résolvent aussi un problème technique posé par l'emploi du langage : le nombre limité de paramètres d'une fonction définie, par la possibilité de regrouper des paramètres disparates en une seule variable, comme le montre l'exemple suivant :

```
    ∇ FONC X;A;B;C
[1]   A←>X[1]
[2]   B←>X[2]
[3]   C←>X[3]
[4]   ...
[25]  ... ∇

      MAT←10 10⍴.5 .3 1.75 ¯2 ...
      NOM←10 6⍴'DUPORTARNAUD ...'
      TAILLE←3

      FONC  (MAT;NOM;TAILLE)
```

Cet artifice se révèle très intéressant dans les problèmes d'interface entre APL et d'autres langages. Les tableaux généralisés résolvent le problème posé par la syntaxe d'appel d'un sous-programme Fortran.

III - OBJECTIFS - APPROCHE

Après considération des propositions existantes, où il est davantage question d'élégance et de régularité du nouveau langage obtenu plutôt que de la faisabilité de sa possible réalisation ou du confort de son emploi, notre souci s'est porté vers la sélection d'un noyau de primitives très simples. Nous nous sommes pour cela fortement inspirés d'une rare réalisation commercialisée (I5), qui reprend elle-même les travaux de Gandhour et Mezeï (T2).

Notre objectif consiste donc à fournir un noyau de primitives de construction de tableaux généralisés, à étendre ou adapter l'action de certaines primitives existantes et à laisser le système ouvert pour de futures extensions.

III.1 - Approche choisie

Nous appellerons objet ou donnée de **base** , les tableaux du langage d'Iverson. Les tableaux généralisés se définissent de manière récurrente à l'aide des données de base.

III.1.1 - Définition : un tableau généralisé conserve une structure rectangulaire classique. Cette structure est définie par la connaissance d'un rang, d'un vecteur de dimension(s) et d'un ensemble ordonné de valeurs : ses éléments; un élément d'un tableau généralisé est soit un objet de base, soit un autre tableau généralisé.

En conséquence, les données de base ne peuvent se trouver qu'aux **feuilles** des arborescences ainsi définies.

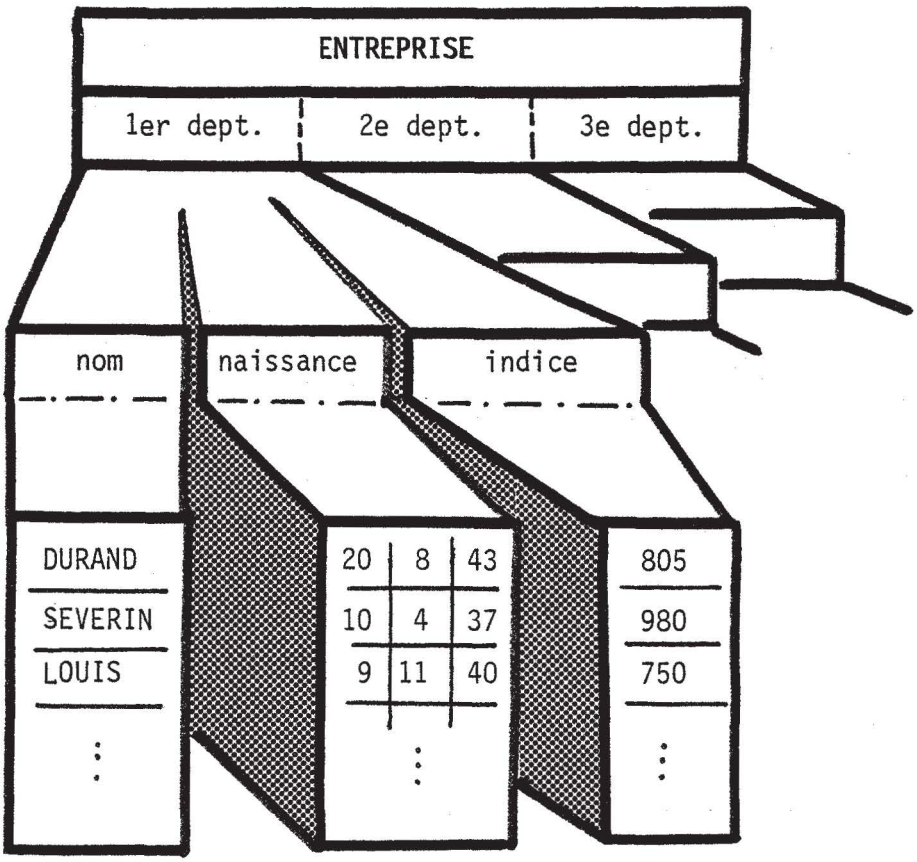


Figure 7 : L'entreprise

III.1.2 - Exemple : Soit un fichier simplifié du personnel d'une entreprise. Une fiche individuelle rassemble les informations :

- NOM(n) : vecteur de n caractères
- DATE(3) : vecteur de 3 entiers
- INDICE : scalaire entier

Trois départements composent l'entreprise, chacun d'eux ayant un nombre variable de personnels. La structure de l'entreprise est bien représentée par le diagramme de la fig.7. L'entreprise apparaît comme un vecteur de 3 éléments de type "département"; un département rassemble trois fichiers conjoints; le fichier des noms est un vecteur de vecteurs de caractères.

III.1.3 - Représentation graphique des tableaux généralisés : Dans la suite de cet exposé, on représentera les tableaux généralisés sous la forme graphique d'arbres (T3). Le tableau, lui-même, est représenté par un noeud (•), ainsi que chaque élément de chaque sous-tableau, avec, à gauche du noeud, le vecteur de dimension(s) de ce noeud. Un élément de base est représenté par les valeurs de ses éléments - numériques ou caractères -. Les branches reliant les noeuds représentent le relation "être le fils de". Ces branches sont ordonnées de la gauche vers la droite en fonction de l'ordre lexicographique de parcours du tableau, pour un même niveau donné.

Scalaire :

```
A←2 ⍝          •
               2
```

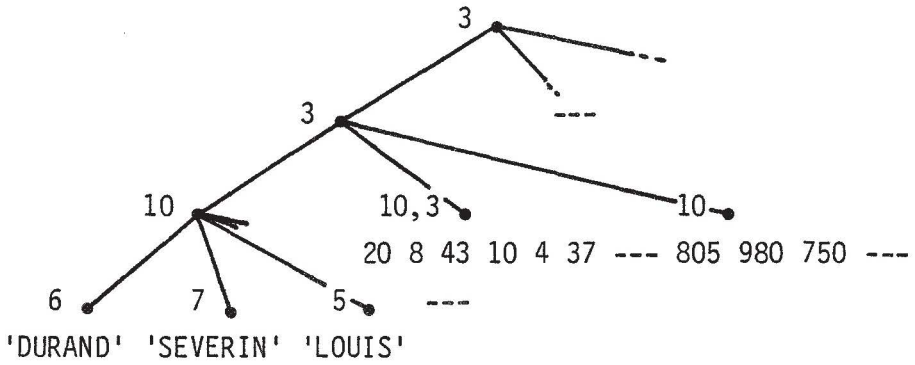
Vecteur :

```
B←11 12 ⍝      2 •
               11 12
```

Matrice :

```
D←2 2⍴'ABC' ⍝ 2,2 •
               'ABCA'
```

L'entreprise :



IV - DEFINITION DU NOYAU MINIMAL

Une fonction de construction d'un scalaire généralisé et les formes étendues des primitives de restructuration (ρ_2) et de concaténation ($,_2$) constituent un noyau générateur. Cependant, à l'usage, très rapidement des formes de construction abrégées et surtout des primitives de sélection nouvelles s'avèrent indispensables.

IV.1 - Primitives de construction

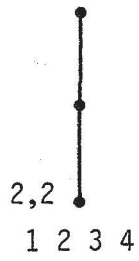
IV.1.1 - Fonction monadique de capture ($<_1$ ou $\subset_1$ dans d'autres systèmes), "enfermer" ou "recouvrir" (enclose,seal,conceal) : elle s'applique à une donnée quelconque , pour créer un scalaire généralisé dont l'unique valeur est cette donnée :

```
A←<'DCAB' ⍝    A :
```

Cette fonction s'applique à un opérande absolument quelconque en type, rang ou taille. Signalons que ce point précis constitue le sujet des premières divergences entre théoriciens, certains d'entre eux préférant par exemple que cette primitive laisse les scalaires invariants.

```
B←<,2 ⍝        B :
```

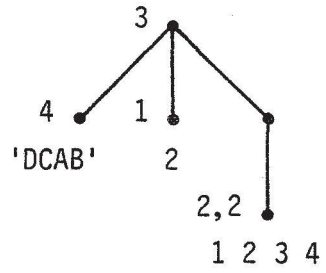

```
C←<<2 2⍴1 4 ⍝  C :
```



Si l'on étend la fonction de concaténation ($,_2$) et celle de restructuration (ρ_2), il devient possible de construire un vecteur de

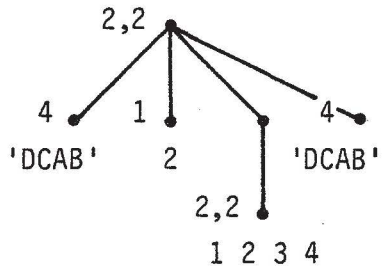
tableaux de base :

```
D←A,B,C      ⍝ D :
```



qui peut servir d'opérande dans une construction :

```
E←2 2⍴D      ⍝ E :
```



IV.1.2 - Fonction monadique de libération ($>_1$ ou $\supset 1$), "découvrir" ou "révéler" (disclose,unseal,reveal) : cette primitive assure la fonction inverse de la précédente, permettant de dégager un niveau de capture. Ainsi :

```
      > A          ⍝ rend l'objet enfermé par A
DCAB
```

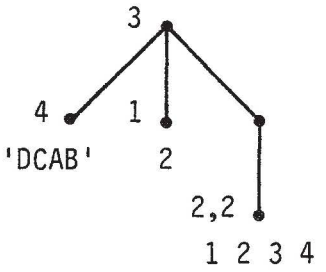
L'opérande doit posséder au moins un niveau de capture et n'avoir qu'un seul élément.

```
      >>C
1  2
3  4
```

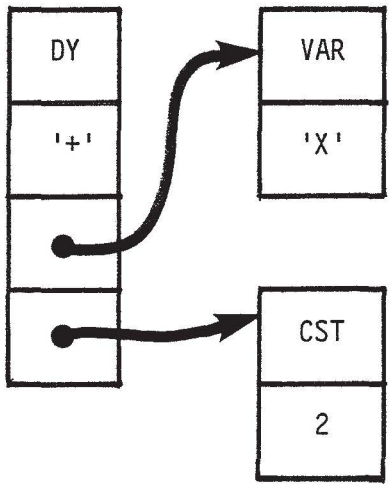
Dans les autres cas, la fonction de libération reste neutre.

```
      >3
3
```

```
F←>D         ⍝ F :
```

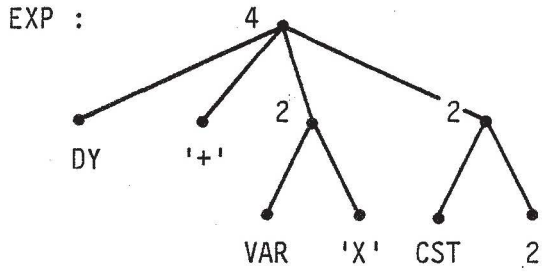


IV.1.3 - Facilités d'écriture : nous présenterons d'abord un exemple tiré de (T11) illustrant l'utilisation des tableaux généralisés en analyse syntaxique. Soit l'expression arithmétique simple 'X+2' que l'on représente par son arbre syntaxique de façon très classique :



dans laquelle DY, VAR, CST sont des entiers caractérisant les types des unités : opérations, variables, constantes. A l'aide des primitives précédentes, on peut maintenant écrire :

$EXP \leftarrow (<DY), (<'+'), (<(<VAR), <'X'), (<(<CST), <2)$ ⍝ (1)



Le moins que l'on puisse dire est que la formulation (1) souffre de lourdeur et surtout d'inexpressivité. Pour y remédier nous introduirons deux nouvelles primitives.

IV.1.4 - Fonctions dyadiques de concaténation : ces deux primitives ont été proposées par Gandhour et Mezeï (T2). Il s'agit de :

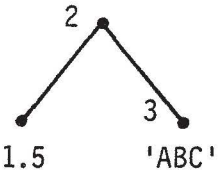
a) la fonction paire $(\bar{,}_2)$ que l'on définit par une équivalence :

```
A,B  ⍝   <=> (<A),(<B)
```

A partir de deux opérandes quelconques la fonction paire fournit un vecteur contenant comme éléments les deux opérandes.

```
G←1.5,'ABC' ⍝
```

G :



b) la fonction chaînage ($\diamond_2$) définie aussi par équivalence :

```
A◇B  ⍝   <=> (<A),B
```

permet de créer des vecteurs plus longs. L'opérande droit doit être un vecteur ou un scalaire structuré. L'association d'une primitive paire et de plusieurs chaînages se fait toujours de la même façon :

```
EXP←DY◇'+'◇(VAR,'X'),(CST,2)  ⍝  (2)
```

(1) et (2) sont donc équivalentes.

IV.1.5 - Abréviation d'écriture et création explicite de listes :

Cette notion de listes se rencontre dans de nombreuses réalisations sous des formes apparemment diverses :

- liste d'index d'une variable :

```
M[I1;I2;I3]
```

- éditions mixtes (en voie de disparition) :

```
'RESULTAT : ';SCORE←3.75
RESULTAT : 3.75
```

- arguments de certaines fonctions système (L9,L10,R4) :

```
'I3,F7.2' ⎕FMT (N;R)
```

Les tableaux généralisés permettent d'unifier ces notations en définissant une liste comme un vecteur d'objets de base par la relation d'équivalence :

$(A_1;A_2;\ldots;A_{N-1};A_N)$ ⍝ <=> $A_1 \diamond A_2 \diamond \ldots \diamond A_{N-1},A_N$

Si nous reprenons notre exemple, (2) se simplifie encore :

```
EXP←(DY;'+';(VAR;'X');(CST;2))  ⍝  (3)
```

Seules les parenthèses utiles garnissent l'expression (3).

D'autres systèmes ont adopté depuis peu une notation encore plus agréable, dite "strand notation" :

```
EXP←DY '+' (VAR 'X') (CST 2)
```

IV.1.6 - Application à l'indexation :

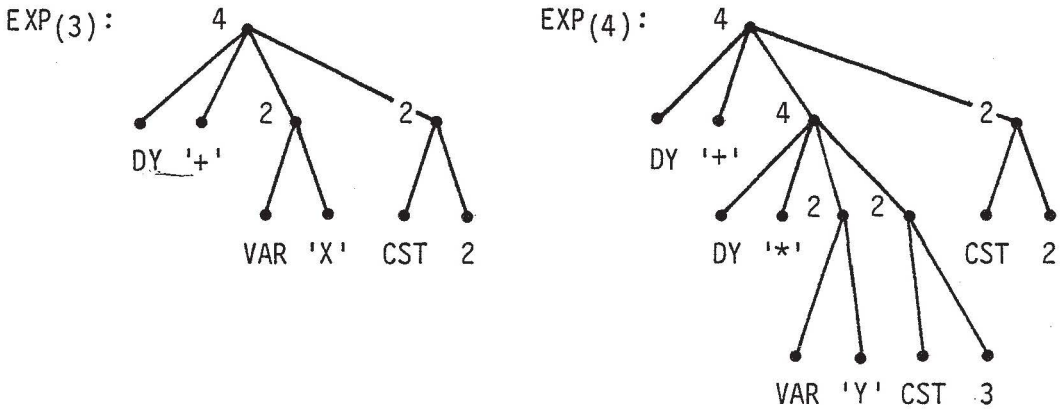
Les écritures suivantes deviennent équivalentes :

```
M[I1;I2;I3]
I←(I1;I2;I3)      puis   M[I]
ou mieux                 I⌷M
```

Il devient alors possible de passer une liste d'index comme paramètre d'une fonction. Les opérations d'indexation acquièrent plus de généralité et surtout leur formulation devient indépendante du rang de l'objet indexé.

Le prolongement de l'affectation indicée simple aux tableaux généralisés offre une primitive très puissante de modification sélective des arborescences.

```
EXP[3]←<(DY;'*';(VAR;'Y');(CST;3))   ⍝   (4)
```



L'expression correspondant à (4) étant : '(Y*3)+2'

Nous étudierons plus précisement les problèmes posés par l'affectation indicée des tableaux généralisés.

V - EXTENSION DES PRIMITIVES SCALAIRES

Dans ce cas une alternative se présente :

- étendre quand cela est possible (homogénéïté de type des feuilles des structures en présence) l'action des primitives scalaires aux nouvelles structures (T4), ce qui suppose une généralisation du principe d'extension des scalaires :

```
X←1÷(1 2;(5;2 2⍴1))
```

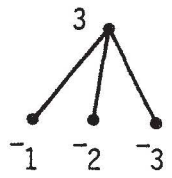
L'opération s'applique à toutes les feuilles. Le scalaire 1 est étendu en "profondeur" à la structure de droite, le résultat sera donc :

```
(1 .5;(.2;2 2⍴1))
```

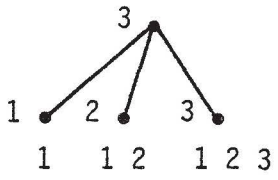
- introduire de nouveaux opérateurs permettant de ne le faire qu'explicitement (T3) au premier niveau d'un tableau, ou à tous les niveaux et récursivement. Cette extension présente plus de généralité et englobe la précédente. Deux opérateurs sont proposés :

V.1 - Opérateur "application itérative" ($\ddot{}_1$)

$B \leftarrow \ddot{} - A \leftarrow (1;2;3)$ ⍝ B :

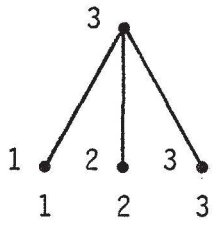


$C \leftarrow \ddot{} \iota A$ ⍝ C :



V.2 - Opérateur "application récursive" ($\ddot{\downarrow}_1$)

$D \leftarrow \ddot{\downarrow} \rho C$ ⍝ D :



Ces concepts très pédagogiques apportent beaucoup à l'utilisateur. Leur réalisation suppose de la machine d'exécution une qualité supplémentaire : l'application itérative ou récursive d'une primitive donnée sur tous les éléments d'un tableau généralisé.

Une simulation en APL étendu présentée en annexe 3 prouve leur faisabilité.

VI - PROBLEMES ET BESOINS

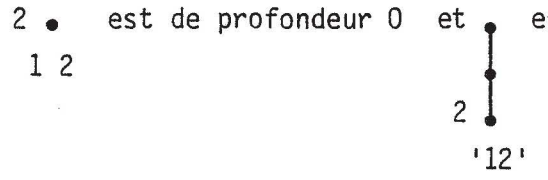
VI.1 - Edition des tableaux généralisés

Le problème est celui de l'édition standard implicite d'un

tableau généralisé, une nouvelle information devant être communiquée à l'utilisateur : la structure du tableau. Dans un premier temps, on peut simplement signaler par un message spécial que l'objet est un tableau généralisé. Nous présentons en annexe-3 une fonction générale d'édition rédigée en APL.

VI.2 - Profondeur

Avec les tableaux généralisés apparaît une nouvelle grandeur caractéristique d'une donnée : sa profondeur ou le nombre maximum de niveaux d'imbrication qu'elle contient. Ainsi :



VI.3 - Indexations généralisées

Directement liés à cette notion de profondeur, des besoins en primitives nouvelles de sélection se font ressentir. L'objectif est de permettre d'atteindre n'importe quel élément d'un tableau et à n'importe quel niveau en précisant simplement quel est le chemin pour le désigner ou l'extraire. La primitive cherchée est en fait la généralisation d'applications de sélection simple de composante et de leur libération.

L'extraction de la chaîne 'Y' du tableau EXP construit en (4) peut s'obtenir par :

```
      >(>(>EXP[3])[3])[2]   ⍝   (5)
Y
```

Cette formulation est bien moins agréable que (6) :

```
EXP⌷(3;2;2)   ⍝   (6)
```

qui rend la même valeur et dans laquelle figure en clair le chemin d'accès voulu. De plus, un tel chemin peut être manipulé par programme :

```
V←(3;3;2)
EXP⌷V
```

VII - EXTENSION DES AUTRES PRIMITIVES

Après cette première étape d'introduction des primitives de construction, il convient de recenser les primitives existantes susceptibles d'extension, et les besoins en nouvelles primitives.

VII.1 - Primitives de sélection

Comme pour la restructuration et la concaténation, il semble naturel d'étendre toutes les primitives de sélection uniformément. Elles s'appliqueront sur le premier niveau d'un tableau généralisé. Nous étendrons donc :

- linéarisation et laminage
- renversement et rotation
- compression et expansion
- transposition simple et généralisée
- prend et laisse
- dimension de

Nous reconduirons l'indexation simple qui est également une primitive de sélection. Par exemple :

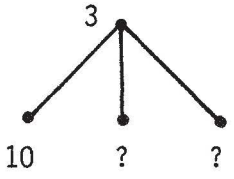
```
      >EXP[2]
   +
```

"Simple" signifie que les indices utilisés restent des données de base, c'est à dire des scalaires, vecteurs ou tableaux à valeurs d'entiers, par opposition à une indexation généralisée où les indices seraient structurés.

VII.2 - Remplissage complétif

Que donnera alors une expression comme ?

```
X←3↑<10   ⍝     X :
```



Dans le cas des nombres, l'élément choisi pour compléter est 0. Pour les caractères, c'est le "blanc". Dans le cas du calcul

précédent, certains préconisent de compléter d'objets vides. D'autres experts vont jusqu'à préserver au delà du vide la forme et le type de la structure constitutrice d'origine; ils définissent alors le **prototype** d'un tableau généralisé (T8,T12). Pour les objets de base, 0 et blanc sont des formes simplifiées de prototype :

```
A←(1 1;2 2;3 3)  ⍝        A :        3
                                    2 2 2
                                    1 1 2 2 3 3

B←0⍴A            ⍝        B :        0 .

C←1↑B            ⍝        C :        1
                                    2
                                    0 0
```

D'un point de vue purement intuitif, la notion de prototype est bien ressentie pour un tableau régulier (dont tous les éléments ont la même structure, comme dans l'exemple précédent). Ainsi, on conçoit bien que pour un fichier - ou vecteur d'articles ou de fiches de même composition - l'élément "à vide" soit une fiche dont tous les champs numériques soient nuls et tous les champs de type caractère remplis de blanc. Cependant, quand la donnée n'est pas régulière, prendre la structure de son premier élément comme prototype semble quelque peu arbitraire.

VII.3 - Primitives utiles

a) Simplicité ($\equiv_1$) :

Cette primitive rend la valeur 1 ou 0 suivant que son opérande est un tableau de base - une donnée simple - ou une structure.

```
      ≡2 2⍴1
0

      ≡<2 2⍴1
1
```

b) Identité (Ξ_2) :

Il est nécessaire de pouvoir comparer deux structures niveau par niveau, et dans chaque niveau terme à terme. La première différence rencontrée déterminera une réponse négative.

c) Appartenance et élément de (ϵ_2 et ι_2) :

Ces deux primitives deviennent facilesà étendre dès l'existence de la précédente et rendront des services comme :

- identification ou recherche d'articles dans un fichier
- extraction de sous-structures vérifiant des critères déterminants et sélection
- tris multicritères

VIII - LA REALISATION

Nous présentons ici comment s'intègrent les tableaux généralisés dans l'architecture de l'interprète. Notre approche est en fait une étape vers les tableaux généralisés, l'interprète manipulant de façon interne un nouveau type de données: le type "structure".

VIII.1 - Objectifs

Dans cette réalisation nous nous efforcerons de surcroît d'atteindre les objectifs suivants :

a) ne pas limiter la taille des feuilles (éléments terminaux)

b) ne limiter ni la profondeur des structures, ni leur nombre d'éléments

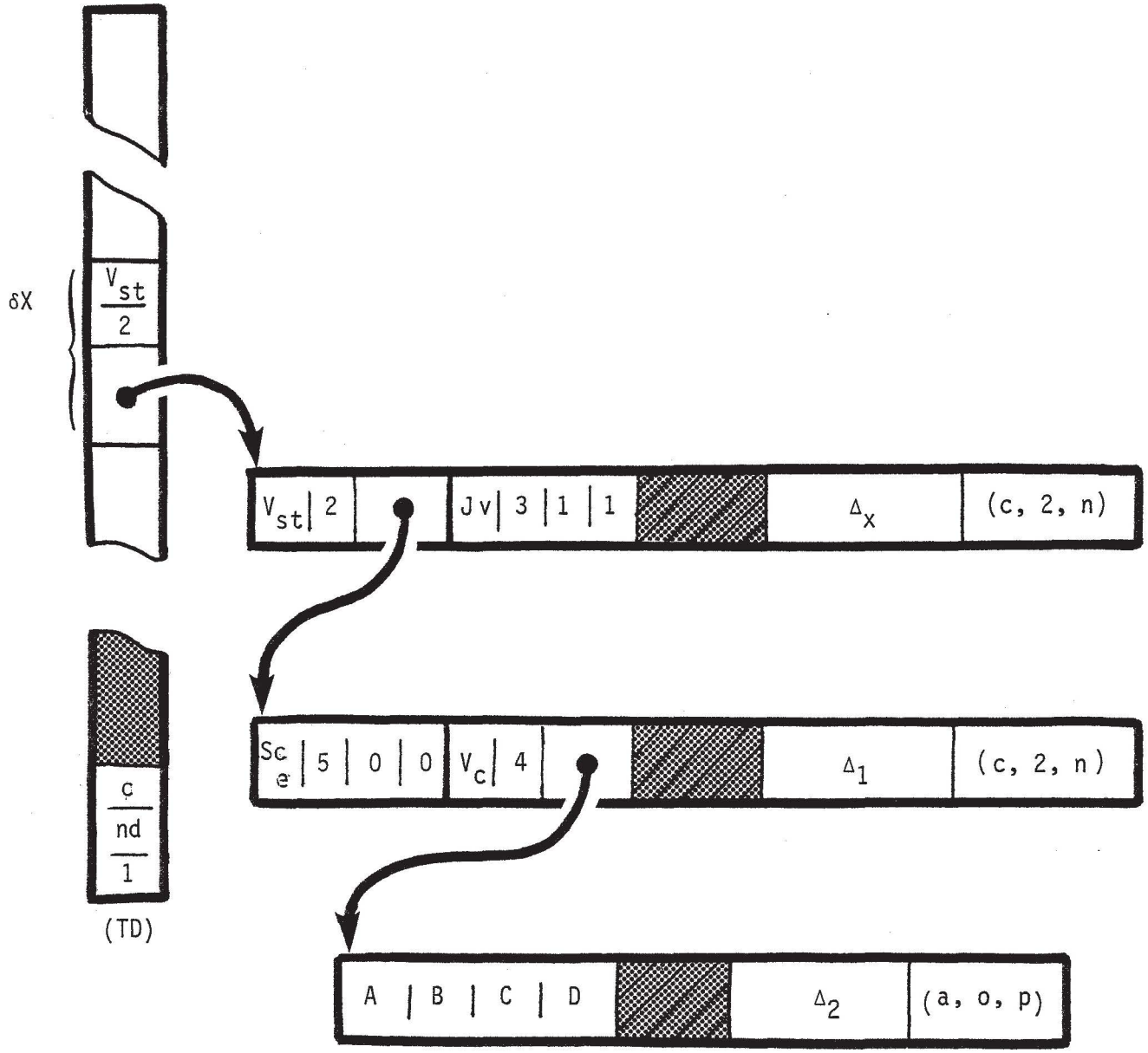
c) obtenir un temps d'accès aux nouveaux objets du même ordre de grandeur que les variables ordinaires

d) faire des tableaux généralisés des variables à part entière de la zone de travail active : variables sauvegardables, effaçables, copiables depuis une zone inactive

e) obtenir de bonnes performances en temps de traitement ainsi qu'une utilisation rationnelle de la mémoire.

MEMOIRE APL (M)

Table des descripteurs



V : Vecteur
Sc : Scalaire
Jv : J-vecteur
st : Structure
e : entier
c : caractère

δ : descripteur primaire
Δ : descripteur complémentaire

Figure 8 : X←((5;'ABCD');⍳3)

VIII.2 - Structure de données adoptée

Nous introduisons un nouveau type d'élément : le type descripteur ou encore type structure. La racine d'un tableau généralisé sera représentée par un bloc de mémoire APL du type "liste de descripteurs" (cf chap.3). Ainsi, la variable X suivante,

```
X←((5;'ABCD');⍳3)
```

aura l'allure décrite à la fig.8. La variable X est un vecteur de type structure, possédant deux éléments. Le premier élément est de type structure : (5;'ABCD'), le second est un objet de base : ⍳3.

Un tableau généralisé est donc représenté par une arborescence de blocs, dont les feuilles sont représentées comme les objets de base, c'est à dire soit par un seul descripteur, comme (⍳3) ou (5), soit par un descripteur pointant un bloc de valeurs, comme ('ABCD').

En créant ce nouveau type structure, nous levons le problème de l'hétérogénéïté des éléments d'un tableau, ce qui satisfait les objectifs (a) et (b).

VIII.2.1 - Inconvénients :

Cette méthode présente les inconvénients suivants :

1°) Prolifération accrue des blocs de mémoire : ce désavantage conduit à la fragmentation de la mémoire, mais il est cependant atténué par plusieurs aspects positifs de la réalisation :

- le partage des données : nous tirerons profit encore une fois des compteurs d'utilisation des blocs, ce qui satisfait l'objectif (e). Les variables créées dans l'expression

```
B←<A←1 2 3
```

seront représentées de façon interne par les blocs de la fig.9.

- les formes auto-décrites - ou objets sans support dans (M), uniquement décrits par un descripteur de (TD) tels le scalaire, le j-vecteur, le vecteur vide -

- la mémoire virtuelle réduit quelque peu cet inconvénient.

2°) Lenteur d'accès : pour des structures profondes, l'accès au feuilles nécessite de traverser tous les niveaux depuis la racine, ce qui peut fortement solliciter les échanges entre mémoire principale et mémoire secondaire

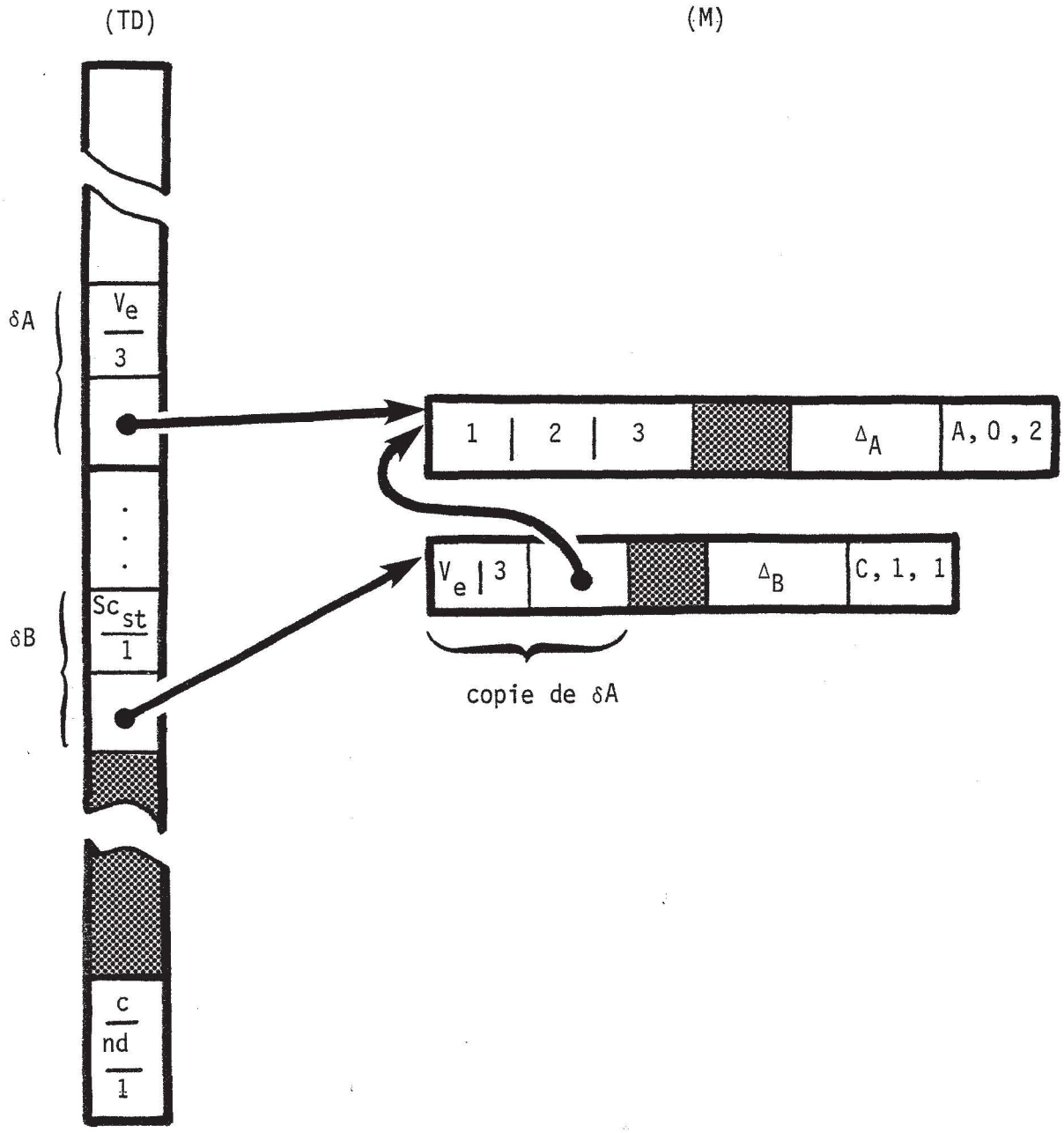


Figure 9 : B ← < A ← 1 2 3

VIII.2.2 - Avantages :

En compensation, elle offre de nombreux avantages.

1°) Les objectifs concernant la taille des feuilles (a) et la profondeur des tableaux généralisés (b) sont atteints.

2°) Contiguïté des éléments par niveau : la racine d'un tableau généralisé a exactement la même organisation que les données de base : ces éléments constitutifs sont contigus, ceci restant vrai à chaque niveau. En conséquence, les algorithmes existant pour la réservation, les accès pourront être utilisés. Les règles d'optimisation présentées au chapitre 3 restent applicables.

3°) Création d'un tableau généralisé : un opérateur, créant un tableau généralisé, ne fabrique que le premier niveau de celui-ci en ne demandant qu'un seul bloc de mémoire (objectif (e)). L'opérateur remplit ensuite ce bloc avec des éléments de type descripteur, comme il sait déjà le faire avec des éléments du type caractère ou numérique. Un descripteur pouvant renfermer un pointeur sur la mémoire APL, son écriture dans un bloc sous-entend un partage qui doit être validé par une duplication logique si l'on veut assurer le maintien de cohérence de la mémoire des blocs. Cette duplication, si nécessaire, peut être générée au moment de l'écriture de l'élément descripteur. Ceci compliquant inutilement les procédures d'accès, nous avons préféré intervenir globalement après la construction du résultat en appliquant la seule règle suivante :

Chaque création d'une nouvelle donnée de type structure doit être suivie d'une application de la primitive de duplication logique sur tous les blocs mémoire pointés par cette donnée. Si A est l'adresse virtuelle de la donnée nouvelle : ⍎ (BDUP,A)

4°) Destruction d'un tableau généralisé : des mêmes propriétés de la gestion de mémoire découle la simplicité de cette destruction : l'effacement d'un tableau généralisé est une primitive existante : BKILL(A)

5°) Sauvegarde : aucune structure de travail additionnelle n'ayant été introduite - table ou autre - la sauvegarde et la restauration de tableaux généralisés est une facilité acquise.

VIII.2.3 - Remarques :

Au cours de cette réalisation, quelques aspects assez intéressants concernant les tableaux généralisés ont été rencontrés.

1°) Copie d'objets arborescents depuis une zone inactive :

La zone de travail active (M) et une zone de travail inactive (M') renferment des objets implantés dans deux espaces virtuels indépendants. Amener un objet de nature arborescente (O') de (M') dans (M) consiste donc à construire un nouvel objet (O) en zone active à l'aide de mémoire puisée dans (M). Ce mécanisme de recopie est manifestement récursif dans le cas des tableaux généralisés.

D'un autre côté, durant le travail de reproduction dans (M) de l'objet (O'), il est souhaitable de récupérer dans (M) les images de partage existant dans (M') pour le compte de l'objet (O'). Ce souci a son importance dans le cas d'une commande de copie globale de la zone inactive qui concerne tous les objets.

La solution que nous avons mise au point consiste à :

- gérer une table temporaire de correspondance (TC) entre les adresses virtuelles de (M') rencontrées au cours de la descente récursive de l'objet (O') et les adresses virtuelles des blocs créés au fur et à mesure pour leur image dans (M)
- reproduire une forme particulière d'exécution de la primitive "copie" sur un bloc d'adresse virtuelle A' : ⊛ (BRECOP,A').

A la rencontre de toute adresse A' de (M'), un balayage de la table d'équivalence (TC) conduit aux actions suivantes :

a) En cas d'échec (A' n'appartient pas à TC) : demande d'un bloc mémoire de taille équivalente dans (M) d'adresse A, introduction dans (TC) de la relation A' $\underline{=}$ A, recopie physique de A' sur A, ⊛ (BRECOP,A') si le bloc A' a des fils

b) En cas de réussite : inscription dans (TC) de l'équivalence A' $\underline{=}$ A, duplication logique dans (M) de A : BDUP(A)

2˚) Affectation indicée :

L'affectation indicée transgresse à la fois les règles syntaxiques et sémantiques premières d'APL. Elle ne consiste pas comme toutes les autres primitives à la création complète d'une nouvelle donnée, mais à la modification sélective de certaines composantes désignées par des indices d'un objet **existant** .

Une telle nuance n'affecte pas le traitement interne de cette opération quand elle s'applique à un objet de base :

```
A←3 2 1
A[2]←5
```

Ici, la valeur du 2^{e} élément de A devient 5, les autres ne changeant pas. En revanche, quand elle concerne un tableau généralisé, la représentation choisie nous pose un problème :

```
B←(1 2 3;'ABC')
B[2]←A←<2 2⍴'XY'
```

Il est clair que nous ne pouvons pas aussi simplement remplacer une valeur de l'indexé (B) par une valeur de l'indexant (A) sans éviter des problèmes de mémoire.

Ce cas précis fait entorse à la régularité de fonctionnement de l'affectation indicée. Nous avions prévu cet algorithme indépendant du type des éléments affectés et du nombre d'indices employés et nous sommes contraints, dans ce cas, d'intervenir à chaque itération de l'opération car, à chaque modification élémentaire, il faut :

- dupliquer logiquement d'abord l'élément indexant courant
- détruire logiquement ensuite l'élément indexé courant

(indexé et indexant peuvent être dépendants l'un de l'autre dans leur réalisation mémoire)

- remplacer enfin l'élément courant de l'indexé par l'élément courant de l'indexant.

3°) Table des descripteurs :

La table des descripteurs est l'ancrage de tous les objets de la zone active. Nous l'avons réalisée dans la mémoire virtuelle comme un tableau généralisé : elle est identifiable au vecteur de tous les objets de la zone active, variables, fonctions définies ou machine...

Si nous appelons Δ l'adresse virtuelle de cette table en zone active (Δ' dans une zone inactive), nous obtenons les résultats suivants :

a) ⍎(BRECOP,Δ') effectue une recopie totale en zone active de tous les objets contenus dans la zone inactive (M'). En fait, il faut également mettre à jour le code machine des fonctions définies et introduire ou mettre à jour les nouveaux noms externes dans la table des symboles.

b) ⍎ (BKILL,Δ) détruit tous les objets de la zone active. Cette constatation nous a conduits à l'introduction d'une commande système utilitaire permettant de tester la cohérence de la mémoire logique. En effet, l'appel de cette commande doit ramener **théoriquement** la zone active à l'état vierge. Cette commande s'avère être un excellent outil de détection des zones de travail endommagées.

IX - CONCLUSION

Nous avons atteint le but que nous nous sommes fixé en respectant dans leur ensemble les objectifs de réalisation supplémentaires que nous avons imposés.

L'introduction des primitives de construction de structures n'a pris que deux ou trois heures en temps de développement complet. Hormis le problème particulier des éléments de remplissage pour les primitives ↑ 2 et \ 2, l'extension des primitives de sélection a été entièrement automatique, seul l'algorithme de l'affectation indicée a été légèrement modifié.

Les réels problèmes sont venus des transferts depuis une zone inactive de tableaux généralisés, un aspect de la question auquel nous n'avions pas accordé suffisamment d'attention au départ, mais auquel nous avons apporté tout de même une solution assez élégante.

L'ensemble constitue un système fonctionnel, très fiable et presque complet.

L' introduction des primitives de construction de structures, l'extension des primitives de sélection et l'adaptation de l'affectation indicée nous a permis de simuler dans l'APL ainsi étendu tous les opérateurs et primitives proposés par Gull et Jenkins (T3) - cf annexe-3 - et d'expérimenter une interface APL-FORTRAN simplifiée.

La possibilité de charger sous notre système des fonctions machine en tant qu'objets de la zone active et l'introduction des tableaux généralisés nous ont permis d'exécuter du code généré par un compilateur Fortran sous le contrôle du système APL.

L'idée consiste à incorporer dans l'interprète, une fonction spéciale assurant d'une part le chargement du code Fortran identique en ce sens à celui d'une fonction machine) et d'autre part la mémorisation de la définition des attributs de ses paramètres d'appel. Cette définition précise permettra de différencier un sous-programme d'une fonction et apportera des renseignements comme :

- existence et type du résultat
- nombre de paramètres formels et pour chacun d'eux le type déposé (compilé par Fortran) et le degré d'interface (paramètre d'entrée, de sortie, d'entrée et de sortie).

L'interprète doit alors reconnaître l'activation d'un tel sous-programme afin d'adapter chacun des paramètres d'appel de façon à ce qu'ils deviennent **accessibles** par le programme compilé. Un travail réciproque doit être accompli au retour du sous-programme pour le ou les résultats. Cette adaptation peut consister à convertir, transposer les tableaux, résoudre des problèmes de résidence en mémoire, annuler les partages mémoire...

Nous présentons en annexe 1 une session de création et de chargement sous APL d'un objet de ce type, puis de son activation.

CHAPITRE 7

INTRODUCTION

Parmi les extensions et améliorations du système APL, une de celles qui nous a paru à la fois intéressante et importante concerne l'introduction d'outils et de moyens de protection des applications utilisant le système APL.

De manière générale, l'utilisateur et le concepteur en attendent :

- efficacité
- simplicité de mise en oeuvre voire automatisme
- finesse de nuancement des protections
- adéquation des nouveaux outils au langage.

Nous étudions d'abord les quelques moyens de protection communs à toutes les réalisations pour démontrer leur pauvreté. Ensuite, après une étude bibliographique des divers efforts qui ont été faits dans ce domaine, nous en ressortons les caractéristiques essentielles de ces systèmes et proposons la solution que nous avons réalisée.

I - OUTILS STANDARD

I.1 - Verrouillage des fonctions

De nombreux éditeurs APL permettent d'ouvrir ou de fermer une fonction définie avec option de verrouillage. A partir de l'instant où une fonction est verrouillée, l'utilisateur ne la voit plus que comme une **boîte noire** . En l'occurence, celle-ci ne pourra plus être ouverte, son contenu devenant secret; une interruption survenant pendant son exécution, sera reportée dans la chronologie des appels à la première fonction appelante non-verrouillée. En conséquence, les variables locales d'une fonction verrouillée deviennent inaccessibles de l'extérieur. Dans les visualisations du vecteur d'états, le système n'imprime plus que le nom de la fonction.

Cette protection est relativement bonne. La fonction verrouillée apparaît comme une primitive du langage, un tout. Une critique est cependant formulable sur sa réalisation dans certains systèmes : des fonctions verrouillées peuvent se retrouver pendantes ou suspendues, ce qui autorise les investigations de leurs variables

locales. La règle la plus stricte est donc, en cas d'erreur, de vider le vecteur d'états jusqu'à ce que plus aucune fonction verrouillée n'y demeure.

I.2 - Protection des zones de travail et des fichiers

L'emploi de mots de passe offre une bonne protection aux utilisateurs et à leurs zones de travail. Un utilisateur choisit un mot de passe qu'il attache à la zone de travail lors de sa sauvegarde. Seule la présentation du mot de passe utilisé à la dernière sauvegarde permettra le chargement de la zone.

Un mot de passe peut être attaché à chaque utilisateur, mot de passe qu'il doit soumettre impérativement au système afin de pouvoir se connecter. Certains systèmes permettent de raffiner les prérogatives des utilisateurs en leur associant en plus un nombre, image de leurs privilèges (R2). En fonction de ce nombre, certaines actions leur sont interdites, charger une zone de travail ne leur appartenant pas par exemple.

Les systèmes munis d'une gestion intégrée de fichiers utilisent encore le mot de passe. A un fichier peut être en plus associée une matrice d'accès qui identifie exactement les utilisateurs autorisés et pour chacun d'eux et de façon très précise les primitives d'accès qu'ils peuvent employer.

Les concepts de mot de passe et de matrice d'accès correspondent en fait à deux philosophies opposées : divulguer à M. X le mot de passe d'une zone de travail équivaut à le communiquer à tout le monde, alors que autoriser M. X à charger cette zone (par une entrée dans une matrice d'accès) est tout à fait différent. La matrice d'accès devrait donc être utilisable également pour protéger les zones de travail.

Voilà résumé le noyau commun à la plupart des systèmes APL. Nous constatons son extrême pauvreté. Le créateur d'applications fermées, écrites en APL, se heurte en effet aux problèmes tels que :

- confier une zone de travail à des utilisateurs divers en rendant tout espionnage impossible et toute interruption transparente
- assurer une sûreté de fonctionnement de ces applications et permettre des reprises automatiques du travail en cas de panne.

Pour tenter d'y remédier, nous nous sommes intéressés à des extensions en matière :

- d'invisibilité des noms
- de dépose de sceau sur les zones de travail
- de sauvegarde automatique sur rupture de ligne
- de contrôle d'accès aux variables
- de recouvrement et gestion des erreurs et interruptions.

II - INVISIBILITE DES NOMS

J.L. Ryan propose une nouvelle commande système qui rend "invisibles" les noms des objets qui y sont précisés (P2) :

)MASK nom1 nom2 ...nomN

Le système remplace alors toutes les représentations externes de ces noms par un symbole conventionnel non-alphabétique (▯). Tout transfert d'objet via les commandes)COPY ou)LOAD transporte ce nouvel attribut de nom. Malheureusement, l'auteur ajoute deux autres commandes :

)UNMASK nom1 nom2 ... nomN

)XLOAD wsid

La première opère le travail inverse, les noms cités redevenant visibles. La seconde permet de charger une zone en filtrant les attributs d'invisibilité des noms.

Nous avons décidé de conserver l'idée de base de Ryan légèrement modifiée. La syntaxe de la commande, au nom près, est identique :

)HIDE nom1 nom2 ... nomN

Les objets, variables ou fonctions deviennent **irréversiblement** invisibles. Ils ne sont plus connus que des fonctions qui y faisaient référence auparavant. Les noms de ces objets sont remplacés dans toute édition par un symbole spécial (⊡). Ils disparaissent du compte-rendu des commandes)VARS,)FNS, etc.

Lorsque l'on cache un objet utilisé dans une fonction, il devient impossible de modifier une ligne où ce nom apparaissait, tout en conservant les références à cet objet. De même, l'objet dont le nom a été caché ne sera plus accessible en mode (⍎) ou en mode (□-entrée), le nom spécifié désignant alors un nouvel objet.

Afin de bien comprendre le mécanisme interne mis en jeu, il faut imaginer que le symbole représentant l'objet ne peut plus être retrouvé dans la table des symboles de la zone. L'objet n'est plus désigné dans les fragments de code machine que par son nom interne (cf chapitre 2). Toute nouvelle codification, due par exemple à l'em-

ploi de ⍎ , de ⍞, ou du mode terminal, recréera pour ce même symbole externe un nouveau nom interne.

Aussi, lorsque l'on cachera des objets pour protéger une application, faudra-t-il prendre garde à ne pas masquer la ou les fonctions principales... Le concepteur de l'application poussera certainement la prudence jusqu'à tenir à jour une version de la zone sans invisibilité.

Exemple :

```
      ∇Z←LIRE
[1]   Z←MYSTERE
[2]   ∇

      ∇ECRIRE X
[1]   MYSTERE←X∇

      ECRIRE ⍳7
      MYSTERE
1  2  3  4  5  6  7
      )HIDE MYSTERE
      ∇ECRIRE[⍞]∇
   ∇  ECRIRE X
[1]   ⍎←X
   ∇
      )VARS
      MYSTERE←⌽'LLYKEJ .RD'
      LIRE
1  2  3  4  5  6  7
      MYSTERE
DR. JEKYLL
```

L'effet de la commande proposée est assez subtil. Deux fonctions de même niveau d'appel peuvent se partager des variables intimes, en les rendant invisibles à l'extérieur. Des noms cachés ne peuvent être spécifiés dans unecommande de copie sélective. Cependant une copie globale de zone de travail transporte tous les objets, qu'ils soient cachés ou non.

III - CONTROLE D'ACCES SUR LES VARIABLES

L'ensemble des efforts de création de nouvelles fonctions de contrôle paraît surtout avoir porté sur les fonctions définies. Il est pourtant intéressant de munir le système de quelques outils de contrôle sur les variables. Citons les idées de T.H. Pucket (P3) introduisant les noms qualifiés, c'est à dire des noms qui ne peuvent

être employés que par l'utilisateur qui les a créés. Pour cela, il qualifie un nom en ajoutant de façon interne au symbole associé le numéro de compte de l'utilisateur.

Les outils disponibles sur les pupitres des unités centrales (arrêt sur instruction, sur référence mémoire) resurgissent sous APL de façon parfois originale et ont inspiré W.J. Maybury (P4) qui propose la **capture** d'une variable. Il emploie pour cela une commande :

)CATCH vname VIA fname

Toute affectation de la variable de nom <vname> déclenchera automatiquement l'exécution de la fonction <fname>. Cette dernière doit être niladique et ne rendre aucun résultat afin de pouvoir intercaler proprement son appel dans l'exécution. On conçoit bien que toutes sortes de contrôles d'accès sophistiqués peuvent être mises en oeuvre grâce à cette commande. La possibilité de définir en APL ces contrôles offre de plus un très grand intérêt.

E.D. Kline, lui, propose d'étendre aux variables les notions de contrôle de trace ou d'arrêt des fonctions (P5) :

V∆vname ← n permet différentes actions, selon la valeur de n, en cas d'affectation de la variable de nom <vname>

Nous avons retenu une solution plus simple et plus répandue (P6,I17) qui consiste à pouvoir verrouiller une variable utilisateur. On peut utiliser une commande)LOCK vname, ou une fonction système ⎕LOCK 'vname'.

La variable devient alors protégée contre les écritures accidentelles. Elle s'identifie à une fonction niladique rendant un résultat.

Exemple:

```
      ALPHABET←⎕AV[137+⍳27]
      ALPHABET
9ABCDEFGHIJKLMNOPQRSTUVWXYZ
      ALPHABET←1↓ALPHABET
      )LOCK ALPHABET
      ALPHABET
ABCDEFGHIJKLMNOPQRSTUVWXYZ
      ALPHABET←ALPHABET,'∆'
SYNTAX ERROR
      ALPHABET←ALPHABET,'∆'
      ^
```

IV - DEPOSE DE SCEAU SUR LES ZONES DE TRAVAIL

APL se singularise par la notion de zone de travail. Assez naturellement une application correspond au moins à une zone de travail. La meilleure façon de proteger une application dans son ensemble est de déposer un **sceau** sur la zone qui la contient. La commande système :

)SEAL

s'avère une des extensions dans ce domaine les plus utilisées. Les réalisations s'entendent même sur les effets d'une telle commande, preuve de sa nécessité.

Le sceau est un attribut attaché à une zone de travail. Cet attribut est transporté dans les sauvegardes et les chargements. La zone est véritablement **gelée** : plus aucune fonction ou variable ne peut être créée ou détruite en mode terminal. Le sceau prend effet immédiatement dans la zone active et il garantit contre les impressions ou dispersions du contenu de la zone. Toutes les fonctions paraissent verrouillées et, ailleurs que dans une fonction préexistante au sceau, les primitives suivantes sont interdites :

- affectation, exécute
- □CR, □EX, □FX
-)COPY,)PCOPY,)ERASE
- les primitives fichiers
- l'ouverture de fonction

Le sceau permet donc la livraison d'applications entièrement fermées, les créations et destructions de variables ou de fonctions restant sous le contrôle total des fonctions fournies par son concepteur.

Exemple :

```
      B←3 3⍴'□□□□o'
      B
□□□
□o□
□□□
      )SEAL
WS SEALED
      A←⍳100
WS SEALED
      A←⍳100
      ∧
      ∇F
WS SEALED
      ∇F
      ∧
      'FILE1' □FCREATE 2
WS SEALED
      'FILE1' □FCREATE 2
              ∧
```

V - CONTROLE ET RECOUVREMENT DES ERREURS ET INTERRUPTIONS

V.1 - Introduction

Une des qualités des langages de haut niveau récents est de fournir à l'utilisateur la possibilité de recouvrir partiellement ou totalement les erreurs susceptibles de survenir pendant l'exécution des programmes. Les précurseurs furent certainement les langages PL1, puis ALGOL 68. ADA, le nouveau langage du Département de la Défense Américaine comportera de telles facilités.

La détection et le recouvrement d'erreur existent déjà de manière standard dans APL : aucune erreur n'est catastrophique. Malheureusement le recouvrement de l'erreur était jusqu'à présent laissé aux soins de la personne opérant sur le terminal.

Ressenties chez tous les utilisateurs comme un besoin, les extensions en ce domaine ne manquent ni en nombre, ni en variété.

B. Legrand pose le problème clairement (L4) :

"Quoi de plus déconcertant, pour une personne qui utilise des programmes écrits en APL et sans connaître le langage, que de voir brutalement apparaître un message LENGTH ERROR ou DOMAIN ERROR? L'erreur peut être imputable à l'utilisateur, au concepteur (qui ne peut avoir tout prévu) voire à des éléments extérieurs, liés à l'ordinateur ou au système APL; quelle que soit la raison, l'utilisateur est dans une impasse. Il ne peut continuer seul s'il ne connaît pas le langage..."

De son côté, le concepteur essaie de protéger l'ensemble de ses programmes des gens qui connaissent trop bien le langage. La touche "escape" ou "attention" constitue dans ce cas le danger le plus redouté. Les efforts se portent donc vers le contrôle de deux sortes d'événements :

- les ruptures d'exécution pour cause d'erreur
- les interruptions volontaires de l'utilisateur pendant les échanges d'entrée ou de sortie ou hors échange.

En général, le contrôle devant être effectué par programme, la forme la mieux adaptée des primitives est la variable ou fonction système. Caractériser depuis un programme la nature et la cause de l'interruption et en définir le recouvrement constitue l'objectif premier de ces extensions. Deux approches sont rencontrées :

- les variables événementielles formelles du type □LX :

elles sont associées à un événement précis (⎕LX, au chargement de la zone qui la renferme) et contiennent une chaîne de caractères représentant une phrase du langage. Cette expression est automatiquement exécutée quand l'événement se produit.

- les trappes : les trappes définissent des points de reprise dans les fonctions définies. Elles sont ou ne sont pas associées à des événement précis.

Après une étude détaillée des propositions ou réalisations existantes, nous exposerons et critiquerons notre approche.

V.2 - Etude des propositions existantes

a) proposition pour APL-SV (1973) :

Le premier effort pour APL, nous le devons à R. Grossman (P7) qui s'inspirant des facilités de recouvrement d'erreurs en PL1 proposa un noyau composé de deux variables système :

- ⎕RR, ou 'report error matrix'
- ⎕RX, ou 'error expression'

Ce noyau répond aux deux problèmes de la caractérisation et du recouvrement de l'erreur. En cas d'erreur, le système APL-SV construit le rapport d'erreur habituel dans une matrice de caractères à l'aide des informations suivantes :

- numéro d'erreur : message d'erreur
- contenu de la ligne erronnée
- indicateur positionnel d'erreur(^)

} ⎕RR

Il exécute ensuite la chaîne de caractère, ⎕RX, comme une phrase du langage : ⍎ ⎕RX.

En espace vierge, ⎕RX initialisée à la chaîne '⎕RR' rend tout le mécanisme transparent à l'utilisateur non averti.

La possibilité de **localiser** ces deux variables dans les fonctions permet de définir des actions différentes selon les fonctions et leur niveau d'appel.

En revanche, tout recouvrement d'erreur doit commencer par une mini-analyse syntaxique, en APL certes, du rapport d'erreur. La propagation des erreurs vers le bon niveau traitant risque de devenir fastidieuse à programmer et peu performante à exécuter.

L'auteur soulève immédiatement le problème des **erreurs récursives** et le résoud partiellement en inhibant les erreurs qui peu-

vent se produire dans l'exécution de l'expression d'erreur, mais non celles qui peuvent arriver dans les fonctions appelées.

b) XEROX-APL (1974) :

W.J. Maybury présente une solution assez originale en définissant le contrôle d'erreur comme un attribut attaché à une fonction définie au même titre que les attributs de trace ou d'arrêt, attachés aux lignes d'une fonction (P1).

Cet attribut est représenté dans le langage par une matrice numérique de deux colonnes. Chaque ligne définit une paire (numéro de ligne, numéro d'erreur traitée) : si une erreur, dont le numéro a été défini dans cette matrice, survient au cours de l'exécution de la fonction, le contrôle est passé à la ligne correspondante dans cette fonction. De plus, il y a dépilement automatique des appels de fonction jusqu'à la rencontre d'une fonction ayant défini une ligne de reprise pour l'erreur traitée et, dans ce cas, c'est la première trouvée qui obtient la main.

Ce traitement des erreurs devient sélectif. Il introduit la notion de ligne de trappe - plus conforme à l'idée de déroutement sur interruption et plus élégant que le remplissage d'une variable système avec une expression de branchement séparée de son contexte. La propagation d'une erreur est automatiquement répercutée à tous les niveaux d'appel jusqu'à rencontre d'une fonction ayant défini une matrice concernée par cette erreur.

c) APLUM (1979) :

La distinction entre erreur et interruption fut introduite par J.H. Burril (P8) qui autorise l'exécution de séquences APL ininterruptibles. Une variable système booléenne ⎕INT permet par son affectation d'inhiber toute erreur ou toute interruption.

d) APL*PLUS (1979) :

Les idées de J.C. Gilmore et de T.H. Pucket (P9) confèrent au système de contrôle d'erreur d'APL*PLUS une très bonne réputation.

Deux variables système localisables, ⎕ALX et ⎕ELX, distinguent les actions à effectuer respectivement en cas d'interruption et en cas d'erreur. Une matrice rapport d'erreur, ⎕DM, construite dès réception de l'erreur par le système fournit un diagnostic exploitable.

Les notions de trappe globale et d'actions spécifiques apparaissent par l'introduction d'une troisième variable système, ⎕SA, qui peut prendre des valeurs conventionnelles comme :

- 'EXIT' : le système dépilera les appels successifs jusqu'à rencontre d'un niveau où ⎕SA ne vaut plus cette valeur
- 'CLEAR' : le système effacera la zone active
- '' : aucune action ne sera entreprise, c'est la valeur en zone vierge.

C'est le retour en mode terminal qui provoque l'action déterminée par ⎕SA. Une application se trouve donc protégée efficacement si, pour une raison inconnue a priori, l'interprète se retrouve en demande d'entrée sur la console APL - qui n'existe d'ailleurs pas forcément -.

e) SHARP-APL (1979) :

Ce système comprend deux variables, ⎕TRAP et ⎕ERR (L10). La première permet de définir localement à une fonction des actions de contrôle très précises. La définition d'une action contient notamment :

- une liste des numéros d'événements concernés par la définition
- un code spécifique d'action (lettre C,E,N,S,D) auquel on peut ajouter une instruction de trappe (chaîne d'exécution).

Les codes d'actions signifient selon leur lettre :

- C (Cut back) : dépiler et exécuter la trappe
- E (Execute) : exécuter seulement la trappe
- N (Next) : passer au niveau suivant
- S (Stop) : retrouver les réactions habituelles du système en cas d'erreur (important pour les mises au point)
- D : option uniquement utilisée en mode terminal pouvant valoir 'EXIT' ou 'CLEAR'.

Les définitions de trappe peuvent être concaténées par l'emploi du délimiteur ∇, elles s'écrivent finalement dans un langage qui n'a plus beaucoup à voir avec APL. Ainsi :

⎕*TRAP* ← '∇2 14 *N* ∇ 0 *E CORRECTION*'

signifie : "à la rencontre de toute erreur, sauf celles répertoriées sous les numéros 2 et 14, appeler la fonction définie sous le nom CORRECTION". Ce système offre tout de même une grande richesse de

de possibilités de définition.

Une fonction monadique, □SIGNAL, permet de faire monter volontairement un événement, ce qui a une grande importance pour la mise au point des applications.

f) APL.S (1979) :

Nous ne terminerons pas ce tour d'horizon sans mentionner ce système décrit par P. Aigrain (P6,P10).

Une fonction système, □TRAP, permet de définir localement une ligne de déroutement dans la fonction qui l'invoque. Une variable système localisable, □ERROR, reçoit le message classique en cas d'erreur et nous retrouvons l'utilisation d'une expression d'erreur, □RX. L'aspect intéressant de ce système réside surtout dans l'ordonnancement des actions entreprises par le système en cas d'erreur :

- construction de □ERROR (si l'erreur n'est pas WS FULL, bien entendu !...)
- exécution de la valeur la plus visible de □RX
- si l'exécution de □RX ne se traduit pas par un branchement, prise en compte de la trappe existante la plus visible (le système dépile les contextes pour rétablir l'exécution au bon niveau)
- édition du message d'erreur et suspension de l'exécution dans tous les autres cas.

V.3 - Enseignements

De l'ensemble de cette étude nous tirons plusieurs enseignements.

1°) Evénements :

La bonne définition du système de contrôle passe par le recensement des événements tels que :

- erreurs d'interprétation
- interruption de l'exécution ou d'un échange
- exécution de certaines actions système particulières telles le chargement ou la sauvegarde de zone de travail, le début ou la fin de session etc.
- le retour en mode terminal.

Le système APL/16 (R2) a défini dans ce but une variable système □CND. Tous les événements susceptibles d'être piégés sont répertoriés par un numéro. Une expression exécutable peut être associée par ce biais à tout événement.

Il est fondamental de pouvoir simuler par programme une montée d'événement.

2°) Portée et sélectivité des trappes :

Dans l'ensemble, tous les systèmes respectent le principe de portée : la validité d'une trappe est étendue également à toutes les fonctions appelées, toute nouvelle définition, consacrée au même événement, prenant le pas sur la définition précédente.

Le système se raffine et devient très souple d'emploi quand on peut définir des trappes sélectives, ces dernières n'interceptant que certains événements en laissant passer les autres.

On retrouve ces deux propriétés dans la définition du langage ADA (G6,G7). Les événements y sont appelés exceptions et les trappes, gérants d'exception.

3°) Trappes globales et actions spécifiques :

Jusqu'à présent nous avons toujours implicitement associé une trappe à une fonction utilisateur. La façon de le réaliser est soit la définition d'une ligne de trappe dans une fonction, soit la localisation d'une variable événementielle et son affectation dans la fonction. Nous pouvons également associer une trappe à la zone de travail active, nous parlerons alors de trappe globale. La recherche d'un gérant d'erreur sera répercutée jusqu'au niveau de la zone de travail, où est appelée la fonction principale.

En général, l'action associée à une trappe est définie par une phrase, ou suite de phrases du langage. Nous dirons que l'action de trappe est explicitement définie. Il est intéressant, et on l'a vu dans le système Sharp, de pouvoir définir des actions implicites, parce que certaines d'entre elles reviennent souvent et parce que toutes ne sont pas forcément possibles ou faciles à exprimer en APL.

4°) Caractérisation et localisation :

Quel que soit l'événement, il faut pouvoir le caractériser et déterminer très précisement son contexte d'exécution. En cas d'erreur, les systèmes s'accordent pour construire une matrice

"rapport d'erreur".

5°) Souplesse d'emploi et intégration au langage :

Le système doit être bien défini, souple d'emploi et harmonieusement intégré au langage. L'utilisation des variables événementielles semble s'y opposer, elle introduit un niveau d'appel parasite dans le vecteur d'états dû à l'appel de l'opérateur ⍎, lequel rajoute une phase de codification nuisible à de bonnes performances.

6°) Récursivité des erreurs :

Dans la mesure du possible et surtout pour rendre moins pénibles les mises au point d'applications utilisant des trappes, le système doit s'efforcer de reconnaître les situations d'erreurs "en boucle".

V.4 - Le système proposé

Le système que nous proposons se compose d'un lot de variables et fonctions système.

V.4.1 - Dernier événement survenu : ⎕EVENT

Cette primitive est une fonction système niladique qui caractérise le dernier événement survenu dans la zone de travail active. Les événements sont répertoriés par classes ; une identification unique leur est attribuée.

R ← ⎕EVENT

Valeur de R	Nature de l'événement
0	Effacement de la zone active
1,2,3	Interruptions
1000 + n	Erreur d'interprétation n° n
2000 + n	Commande système n° n
3000 + n	Anomalie système n° n

La reconnaissance d'un événement survenu est une opération très simple, un test. Il faudrait cependant compléter cette information d'une matrice d'erreur.

V.4.2 - Expression d'erreur : □RX

Cette variable système contient une chaîne de caractères qui sera exécutée seulement en cas d'erreur. Elle est localisable.

Nous ne l'avons introduite que dans un souci de compatibilité avec d'autres systèmes existants.

V.4.3 - Fonction de définition de trappe : □TRAP

Nous introduisons □TRAP comme une pseudo-variable système affectable. Elle permet de définir dans une fonction une ligne de déroutement et dans une zone de travail une trappe globale.

Plusieurs cas peuvent se produire :

1°) Trappes de fonction : □TRAP ← n avec $0 \leq n \leq 999$

N doit être un scalaire entier compris entre 0 et 999. Si l'affectation a lieu à l'intérieur d'une fonction, n représente un numéro de ligne relatif à cette fonction. Si l'affectation n'est pas effectuée par une fonction, alors ce numéro concerne une ligne de la fonction la plus récemment interrompue - au sommet du vecteur d'états .

L'attribution à □TRAP d'une valeur non nulle n implique qu'en cas d'erreur ou d'interruption, la fonction ne sera pas suspendue, mais le contrôle lui sera donné à la ligne n, comme si un branchement à cette ligne avait naturellement eu lieu.

□TRAP peut recevoir à tout moment une valeur nulle, ce qui supprime l'effet de déroutement pour le niveau où a lieu l'affectation.

En cas d'interruption, lorsque plusieurs fonctions apparaissent dans le vecteur d'états, le système réagit de la façon suivante :

- aucune de ces fonctions n'a défini de trappe : il y a impression du rapport d'erreur ou d'interruption et suspension de l'exécution. On retrouve le comportement habituel du système.

- une fonction a positionné une trappe : le vecteur d'états est dépilé afin de retrouver à son sommet la fonction qui a positionné □TRAP à une valeur non nulle.

Le branchement est possible puisque l'on se retrouve dans le contexte d'exécution de la fonction invocatrice.

2°) Actions spécifiques : □TRAP ← n avec ¯6 ≤ n ≤ ¯1

La réaction du système est la même que précédemment : retour par le vecteur d'états à la première fonction ayant définie une trappe, mais suivant la valeur de n, une action spécifique est prise :

Valeur de n	Action dans la zone de travail
-1	"CLEAR" : nettoyage complet de l'espace actif
-2	"EXIT" : sortie de la fonction fautive, ou de la suite de fonctions, jusqu'à et y compris la fonction trappante
-3	"RESET" : interruption d'exécution et vidage complet du vecteur d'états
-4	"NEXT" : passage du contrôle à la ligne suivant le positionnement de la trappe
-5	"ERROR" : Impression effective du compte rendu reportée à la ligne courante de la fonction
-6	"ERROR1": Impression du compte rendu reportée à la ligne d'appel de la fonction

Ce système peut supporter d'autres extensions et assure déjà des protections efficaces.

Lorsque l'affectation de □TRAP a lieu alors qu'aucune fonction n'est active, l'action indiquée ne peut être que spécifique et concerne toute erreur ou interruption survenant dans l'espace actif : la trappe est globale.

La valeur de la trappe peut aussi être connue ou récupérée :

R←⎕TRAP fournit la valeur de trappe du contexte de la demande.

V.4.4 - Montée d'événement : ⎕ERROR

Cette fonction permet au concepteur de faire monter ses propres événements ou erreurs, mais aussi ceux du système. Pour cela deux formes d'appel existent :

```
⎕ERROR  1004

⎕ERROR 'MATRICE NON-REGULIERE'
```

La fonction système agit exactement comme si l'événement avait réellement eu lieu, mais au préalable, elle dépile l'appel de la fonction appelante. L'existence d'une variable ⎕RX ou d'une trappe au niveau supérieur décidera de la suite de l'exécution, en application des règles précédentes. Ce mécanisme permet la propagation des erreurs vers le niveau du gérant.

Quand aucune trappe n'est définie, le rapport d'erreur apparaît au niveau de la fonction appelante.

Exemple :

```
    ∇ R←INVERSE A
[1]   ⍝ CALCUL DE L'INVERSE GENERALISE DE A
[2]     ...
[34]    ⎕ERROR 'MATRICE NON-REGULIERE'
[35]    ...
    ∇

      INVERSE 3 3⍴⍳10
MATRICE NON-REGULIERE
      INVERSE 3 3⍴⍳10
      ∧
```

V.4.5 - Applications :

Les possibilités offertes par ce système sont nombreuses, citons :

- messages d'erreur spécialisés (en langue française par exemple)
- création de sections ininterruptibles
- correction automatique d'erreurs
- installation de niveaux moniteurs dans les fonctions d'une application

- suppression de tests coûteux en entrée de données

- gestion automatique d'erreurs, sauvegardes automatiques et mise à jour de fichiers de surveillance

- gestion des pertes de ligne

- comparativement aux systèmes d'interruption des unités centrales d'ordinateur, il devient possible de simuler en langage APL des primitives nouvelles. Il faut pour cela piéger l'erreur "primitive inconnue", fournir à la fonction APL traitante une copie des paramètres ainsi qu'une représentation de la fonction en erreur et lui permettre de construire un résultat. Des variables système peuvent être créées pour cela, □F, □RA, □LA, □RES ou fonction en erreur, argument droit, argument gauche et résultat. Un tel mécanisme doit être intégré à la machine d'exécution.

V.4.6 - Critiques :

Plusieurs critiques peuvent être émises conformément à l'étude des systèmes existants que nous avons faite.

- les trappes ne sont pas sélectives et seules, pour l'instant, les erreurs sont trappables. En compensation, un positionnement de trappe est une opération très claire et très simple.

- les caractérisations d'erreur sont possibles par la consultation de □EVENT, mais une matrice d'erreur donnerait le nom, le numéro de ligne et la position de l'unité en erreur dans la ligne.

- notre système ne gère pas encore les déroutements sur interruptions externes ou sur retour en mode terminal.

V.4.7 - Illustration :

Pour illustrer l'utilisation de notre système, nous présentons une adaptation en APL d'un programme tiré de (G6) rédigé en ADA (fig.10 et 11).

```
Procedure P is
    ERROR: exception;
    Procedure Q is
        begin
            ...             -- situation 3
            R;
            ...             -- situation 4
        exception
            ...
            when ERROR =    -- handler E2
            ...
    end Q;
    Procedure R is
        begin
            ...             -- situation 5
            ...             -- situation 6
    end R;
    begin
        ...                 -- situation 1
        Q;
        ...                 -- situation 2
        exception
            ...
            when ERROR =    -- handler E1
            ...
end P.
```

Figure 10 : Exceptions d'ADA.

```
    ∇  CAS N
[1]    ⎕TRAP←0
[2]   ⍝ APPEL DE P
[3]    P
    ∇

    ∇  EVENT I
[1]    →0×⍳I≠N
[2]    (IDρ' '),'MONTEE DE L''EXCEPTION'
[3]    ⎕ERROR 0
[4]    'DEROUTEMENT...'
    ∇

    ∇  P;ID
[1]    ID←0
[2]    ⎕TRAP←E1
[3]    'ENTREE DANS P' ◇ EVENT 1 ◇ 'APPEL DE Q'
[4]    Q
[5]    'RETOUR DE Q' ◇ EVENT 2
[6]    →SORTIE
[7]   E1: 'GESTION DE L''EXCEPTION DANS P'
[8]   SORTIE: 'SORTIE DE P'
    ∇

    ∇  Q;ID
[1]    ID←3
[2]    ⎕TRAP←E2
[3]    '   ENTREE DANS Q' ◇ EVENT 3 ◇ '   APPEL DE R'
[4]    R
[5]    '   RETOUR DE R' ◇ EVENT 4
[6]    →SORTIE
[7]   E2: '   GESTION DE L''EXCEPTION DANS Q'
[8]   SORTIE: '   SORTIE DE Q'
    ∇

    ∇  R;ID
[1]    ID←6
[2]    '      ENTREE DANS R' ◇ EVENT 5 ◇ '      CORPS DE R'
[3]    EVENT 6 ◇ '      SORTIE DE R'
    ∇
```

Figure 11 : Listage de la simulation APL du programme ADA précédent.

Cas habituel : aucune exception n'interrompt l'exécution :

```
      CAS 0
ENTREE DANS P
APPEL DE Q
   ENTREE DANS Q
   APPEL DE R
      ENTREE DANS R
      CORPS DE R
      SORTIE DE R
   RETOUR DE R
   SORTIE DE Q
RETOUR DE Q
SORTIE DE P
```

1er cas : L'événement se produit dans P :

```
      CAS 1
ENTREE DANS P
MONTEE DE L'EXCEPTION
GESTION DE L'EXCEPTION DANS P
SORTIE DE P
```

2eme cas : Il arrive dans Q :

```
      CAS 3
ENTREE DANS P
APPEL DE Q
   ENTREE DANS Q
   MONTEE DE L'EXCEPTION
   GESTION DE L'EXCEPTION DANS Q
   SORTIE DE Q
RETOUR DE Q
SORTIE DE P
```

3eme cas : Il survient dans R :

```
      CAS 5
ENTREE DANS P
APPEL DE Q
   ENTREE DANS Q
   APPEL DE R
      ENTREE DANS R
      MONTEE DE L'EXCEPTION
   GESTION DE L'EXCEPTION DANS Q
   SORTIE DE Q
RETOUR DE Q
SORTIE DE P
```

Figure 11 : (suite)

CONCLUSION

Nous avons cherché, dans les pages précédentes, à présenter quelques uns des aspects les plus intéressants de la conception et de la réalisation de l'interprète APL sur le P857.

D'un point de vue théorique, nous avons essayé de décrire le mieux possible l'état de la recherche dans le domaine de l'interprétation du langage APL ; cette description a nécessité un travail bibliographique important mais non exhaustif.

Nous avons ensuite tenté d'approfondir des techniques d'optimisation jusqu'alors principalement utilisées sur de gros ordinateurs (j-vecteurs, algorithmes spécialisés, représentations multiples, beating, etc.) et nous les avons effectivement intégrées dans une véritable réalisation.

Pour y parvenir, nous avons dû nous pencher très attentivement sur les problèmes d'accès aux objets APL lors de l'interprétation. Cette étude théorique nous a permis de découvrir que la nature des opérateurs APL joue un rôle très important dans ces accès et de dégager quatre grands types de parcours, pour la majorité desquels des optimisations ont été apportées. Cette même étude prolongée aux variables optimisées, pour lesquelles nous pensions utiliser a priori un mode d'adressage particulier, a encore permis d'unifier la fonction d'accès aux objets et d'harmoniser ainsi des techniques divergentes.

Enfin, la conception d'une mémoire de partage, initialement prévue pour réduire les consommations mémoire et pour mettre en oeuvre le beating, nous a donné l'occasion de définir une hiérarchie très stricte entre la gestion de la mémoire virtuelle brute d'un côté

et celle des blocs partageables de l'autre. Nous avons généralisé cette gestion de mémoire à deux niveaux à celle automatique des relations de pointeurs entre blocs arborescents, en répertoriant en quatre grandes catégories les structures de données que manipule nécessairement tout interprète APL. De surcroît, une telle gestion s'est révélée extrêmement efficace et d'un emploi très agréable tout au long du développement du système. Par exemple, l'introduction des tableaux généralisés n'a demandé que quelques heures de mise au point.

D'un point de vue pratique, nous pensons également avoir rempli complètement les objectifs de réalisation que nous nous sommes fixés au départ.

1) Qualité :

Le système APL/857 a été certifié et qualifié par une société de service spécialisée, la G.F.I. Automatique, qui a développé et mis au point dans ce but, un logiciel de contrôle et d'évaluation du système APL/857 (A20). Dans son rapport final de qualification, cette société écrit, au sujet du système (A19) :

- quant à sa fonctionnalité :

" APL/857 est un système complet relatif au noyau standard APL. Cela signifie que toutes les fonctions primitives appartenant à ce noyau ont été implantées et fonctionnent correctement. APL/857 se conforme dans l'ensemble à la "pseudo-norme" définie dans la publication de A.D. Falkoff et D.L. Orth, à Rochester, en 1979 (L1) ... "

- quant à son efficacité :

" Les routines sémantiques ont été réalisées de façon extrêmement efficace : les mesures effectuées suivant la méthode de "Harris" donnent des résultats qui placent Philips en très bonne position dans la gamme des mini-ordinateurs. La machine d'exécution au contraire semble extrêmement lente, étant donné que nos mesures ont été effectuées en mono-utilisateur sur une machine dédiée et en utilisant de très petits espaces de travail ... "

Enfin, elle conclut :

" Les critiques précédentes n'entament pas notre jugement d'ensemble : APL/857 est un produit de qualité qui, après amélioration de la machine d'exécution, amélioration de la compatibilité avec I.B.M. et S.H.A.R.P. et introduction de bibliothèques publiques contenant les

utilitaires standard, doit être un argument commercial de valeur pour la diffusion du matériel Philips ... "

2) Mesures et performances :

Sans contester globalement les résultats et les appréciations précédentes, nous y apporterons cependant quelques remarques :

a) La précision de l'horloge utilisée étant relativement faible compte tenu des temps mesurés (cadencée par la fréquence du secteur, celle-ci envoie un top d'horloge toutes les 20 ms), il est dans la plupart des cas nécessaire d'effectuer plusieurs dizaines d'itérations sur l'exécution d'un opérateur donné, pour obtenir une mesure statistiquement correcte. Par exemple, l'équipe de qualification a testé certains opérateurs sur une itération !

b) La fonction système utilisée pour les mesures (□AI) ne donne en réalité que le temps d'allocation de l'unité centrale à l'interprète, sans tenir compte de celui dépensé dans le superviseur, qui, dans une version monoconsole, s'y rajoute obligatoirement du point de vue de l'utilisateur.

Nous avons donc été conduits à reprendre les fonctions de mesure d'une part, en permettant un nombre quelconque d'itérations, et d'autre part, en utilisant la fonction système (□TS), qui donne le temps de connexion, quantité beaucoup plus significative pour un utilisateur assis devant une console.

En ce qui concerne les routines sémantiques, les temps obtenus alors s'avèrent en général légèrement inférieurs à ceux trouvés par G.F.I., et ceci, en dépit de l'aspect pourtant contradictoire des remarques a et b. Ces mesures sont consignées dans le tableau de la figure 13.

En ce qui concerne la machine d'exécution, nous avons reconnu le bien-fondé de l'appréciation de G.F.I., car nos propres mesures ont confirmé une effective lenteur dans la conduite de l'interprétation.

Nous en avons cherché la raison par la mise en place d'un système de collecte de statistiques au niveau du superviseur et constaté que cette lenteur était essentiellement due à une mauvaise répartition du traitement des opérateurs dans l'arbre de recouvrement de la machine d'exécution : en effet, dans plus de 70% des cas, les opérateurs utilisés par une application très courante correspondent soit à des fonctions scalaires, soit à des fonctions de restructura-

MESURES COMPARATIVES DE DIFFERENTS SYSTEMES APL

(Provenance : GFI automatique et Harris Corporation)

Temps d'unité centrale en millisecondes

N° DU TEST	EXPRESSIONS	PH 857	HP 3000	HAR 5123	DEC 2020	CDC CYBER 73	IBM 370 158	HB 66
1	Z←+/VI	33,7	50,6	1,9	3,5	2,0	0,9	13,0
2	Z←∨/VL	3,4	5,4	0,7	6,8	0,5	0,5	2,0
3	Z←⌈/[1]MI	19,6	39,3	2,3	5,3	1,7	1,1	14,0
4	Z←VI*.5	112,4	654,7	57,2	103,9	40,4	42,7	151,0
5	Z←|VR	27,8	101,7	3,6	8,1	1,8	3,5	5,0
6	Z←VR|VI[⍳20]	19,2	34,4	5,3	5,4	4,5	1,1	9,0
7	Z←VI[⍒VI]	432,2	288,1	52,9	56,0	40,4	14,7	135,0
8	Z←¯2 1↑MR	11,4	5,1	1,0	2,8	0,9	0,9	5,0
9	Z←VI∈VI	3534,0	3871,0	38,9	504,3	263,6	61,5	146,0
10	Z←2 1⌽MC	14,4	11,9	138,5	82,1	11,3	4,2	53,0
11	Z←VC∘.=VC	92,6	306,9	21,1	35,8	7,0	5,7	43,0
12	Z←(⍳50)∘.+⍳50	498,8	1193,8	53,9	54,6	37,1	22,3	287,0
13	Z←VR⌊.+VR	55,8	95,3	7,0	12,3	6,6	4,1	35,0
14	Z←MR⊖10↑VR	non ex.	104,3	10,0	33,6	6,4	5,2	11,0
15	Z←1 1 L:→(100>ρZ←Z,+/¯2↑Z)/L	4306,8	2373,4	349,4	1534,6	319,0	196,7	non exécutable

```
VL←1=1 0 1 1 0 0 0 1
MI←10 10ρ VI←(500ρ0 1 0 0 1)/⍳500
MR←10 10ρ VR←VI+.1
MC←26 26ρ VC←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Figure 13 : Quelques mesures

tion. Or, il s'avère que nous avions disposé, pour des raisons de fonctionnalité commune, ces deux catégories dans des branches s'excluant mutuellement de l'espace adressable.

Une réorganisation des modules sources a permis de rendre pratiquement "résidentes" les opérations ou fonctions de l'interprétation les plus couramment sollicitées par l'exécution d'une ligne APL ou même, et surtout, d'une fonction définie.

Dans un deuxième temps, constatant le nombre élevé des demandes de chargement de page, nous nous sommes attachés à réduire, au niveau du superviseur, le temps d'exécution nécessaire aux changements de configuration de mémoire demandés par l'interprète.

Enfin, pour accélérer les accès temporaires aux grosses données, une table inverse a été introduite, qui simule en quelque sorte un ensemble de registres associatifs, et permet de savoir immédiatement si une page virtuelle donnée est présente dans l'espace logique adressable et à quel niveau.

Ces modifications ont permis de réduire de près de 50% les temps d'exécution totaux (interprète et superviseur) pour des applications moyennes.

Signalons qu'aucune modification n'a été apportée aux traitements particuliers des opérateurs et fonctions, ni à la logique de la machine d'exécution.

3) Fiabilité :

Il va de soi que nous n'essaierons pas de démontrer que notre système APL est correct, la preuve d'un tel programme étant hors de la portée de quiconque, du moins dans l'état actuel de la science informatique. Du moins, aurons-nous essayé de formuler un certain nombre d'assertions sur les différents algorithmes utilisés.

En premier lieu, les principaux algorithmes du noyau de l'interprète ont été d'abord validés en APL ou en FORTRAN, sous une forme voisine de leur formulation définitive. Nous avons obtenu ainsi une "machine de base" qui n'a plus guère évolué depuis.

Ensuite, nous avons disposé dans les modules de l'interprète des séquences d'instructions permettant de vérifier que les données restent bien conformes aux assertions.

Enfin, tous les programmes constituant l'interprète s'exécutent dans une mémoire virtuelle, que le superviseur protège contre

les écritures. Cette apparente contrainte de programmation nécessite de respecter un partage constant entre données et programmes mais garantit la réentrance de l'interprète et, en outre, s'est avérée un investissement fructueux pendant la phase de mise au point.

Au cours de cette dernière, un certain nombre d'applications ont été développées par d'autres équipes de l'école.

La plus spectaculaire et la plus probante quant à la fiabilité du système, reste sans doute un compilateur d'un sous-ensemble du langage PASCAL, écrit entièrement en APL/857, par deux élèves de troisième année, H. Bouvet et P. Dussud. Ce compilateur génère un P-code numérique qui peut ensuite être émulé par une P-machine, écrite elle-même en FORTRAN. Le projet complet a été mené à terme, sans que jamais la moindre panne, matérielle ou logicielle, n'interrompe sa progression : un second compilateur, facilement ré-écrit dans ce PASCAL restreint compte tenu des qualités similaires des deux langages, a été entièrement compilé sous APL, puis recompilé sous lui-même et progressivement étendu.

4) Extensions :

Deux des extensions prévues ont été effectivement réalisées, les tableaux généralisés et les outils de protection des applications. La société G.F.I. les a également qualifiées et a jugé leur fonctionnement correct.

Cependant, l'approche que nous avons faite des tableaux généralisés et que nous avons voulue compatible avec une autre réalisation française sur MITRA-15 (I5) devra certainement être révisée, en fonction des résultats de la discussion qui se poursuit dans ce domaine depuis de nombreuses années.

Enfin, à l'heure actuelle, le système est muni d'une gestion de fichiers à composantes, car, satisfaite du produit obtenu, la société Philips a décidé de présenter son système sur un des stands du S.I.C.O.B. 81 et de reconduire auprès de nous ce travail d'aménagement.

En conclusion, nous pouvons donc dire que le but de notre travail, doter un petit ordinateur d'un système APL fiable, étendu, et particulièrement puissant, a été largement atteint.

B I B L I O G R A P H I E

BIBLIOGRAPHIE

1 - GRANDS CONGRES APL

(C71) Colloque APL. 1e et 2e parties, Paris, IRIA, 9-10 Sept. 1971.

(C73) Proceeding of the APL congress 73. Copenhagen, Denmark, August 22-24, 1973.

(C74) Proceedings of the sixth international APL users' conference. May 14-17, 1974, Anaheim.

(C75) APL 75. 11-13 June 1975, Faculty of Engineering, University of Pisa, Pisa, Italy.

(C76) APL76 conference proceedings. Ottawa, 22,23,24 Sept. 1976.

(C79) APL79 conference proceedings. Rochester, New York, U.S.A., APL Quote Quad, vol.9, n°4, part 1 & 2, June 1979.

(C80) APL users meeting. Toronto, October 6,7,8 1980.

2 - PRESENTATION ET DESCRIPTION DU LANGAGE

(L1) IVERSON K.E. A programming language. New-York, Wiley, 1962.

(L2) IVERSON K.E. Algebra as a language. In C71, pp.5-16.

(L3) ROBINET B. Le langage APL. Paris, Technip, 1971.

(L4) LEGRAND B. Apprendre et appliquer le langage APL. Masson, 1979.

(L5) POMMIER S. Introduction au langage APL. Dunod, Paris, 1978.

(L6) FALKOFF A.D., IVERSON K.E. The design of APL. IBM J. Res. Develop., vol.17, n°4, July 1973.

(L7) FALKOFF A.D., IVERSON K.E. APL : manuel de référence, traduit par Pinta et Leborgne, IBM, GHF2-0056, 1973.

(L8) FALKOFF A.D., IVERSON K.E. APL-SV user's manual. IBM, SH-1460, 1973.

(L9) NAKACHE M. Manuel de référence APL. Saint-Etienne, Ecole des Mines, 1976.

(L10) BERRY P. Sharp APL reference manual. I.P. Sharp Ass., March 1979.

(L11) FALKOFF A.D., ORTH D.L. Development of an APL standard. In C79, pp.409-453.

3 - UTILISATIONS D'APL

(U1) RAYNAUD Y. APL, son implantation, son utilisation pour l'aide à la conception des systèmes de traitement de l'information. Bulletin de l'IRIA, Mars 1971.

(U2) DUBRULLE A.A. APL functions for precision control in floating-point computations. In C75, pp.101-114.

(U3) WATSON D. Question : "Why does APL/360/370 give the following results ?". In APL Quote Quad, VOL.5, n°4, 1974, pp.69-75.

(U4) JAWORSKY A. EXEL-APL : un APL structuré. Applications d'APL en France, Congrès AFCET, Juin 1977.

(U5) PAKIN S., POLIVKA R.P. APL : the language and its usage. Prentice Hall, Englewood Cliffs, New Jersey, 1975.

4 - REALISATIONS EXISTANTES

(R1) MARTIN H., LEVY E., RAYNAUD Y. APL : 1e partie : le langage. 2e partie : l'interprétation. Thèse de Docteur-Ingénieur, Toulouse, Université P. Sabatier, Mai 1972.

(R2) GIRARDOT J.J., MIREAUX F. Réalisation d'un interprète complet du langage APL sur un miniordinateur. Thèse de Docteur-Ingénieur, Université de Nancy I, Sept. 1976.

(R3) CHOMAT P. Notions sur les systèmes APL/360. Grenoble, IUT Informatique, Sept. 1971.

(R4) SLIGOS APL80. Sligos, Puteaux, Janv. 1976.

(R5) IVERSON E.B. APL/4004 implementation. In C73, pp.231-236.

5 - METHODES D'INTERPRETATION, DE COMPILATION OU D'OPTIMISATION

(I1) BATTAREL G., DELBREIL M., KALFON P. Un interpréteur APL avec génération de code et réutilisation de code machine. In C71, pp.383-402.

(I2) BATTAREL G., DELBREIL M., TUSERA D. Interpréteur APL optimisé. Rapport n°32, IRIA, Laboria, 1973.

(I3) BATTAREL G. Interprétation optimisée d'APL. 1e partie : structure de l'interpréteur. Thèse de Docteur-Ingénieur, Université de Toulouse III, Mars 1973.

(I4) BATTAREL G., DELBREIL M., KALFON P. I.S.A.A.C. : interprétation sélective d'APL avec compilation. IRIA, Rocquencourt, (1973).

(I5) NEVIANS J. Concept de mémoire virtuelle, optimisation, notion de structure. Thèse de 3e cycle, Université de Paris Sud, Centre d'Orsay, Juin 1975.

(I6) ABRAMS P.S. An APL machine. Ph.D. Thesis, Standford University, Feb. 1970.

(I7) PERLIS A.J., RUGABER S. Programming with idioms in APL. In C79, pp.232-235.

(I8) BERNECKY B. Speeding up dyadic iota and dyadic epsilon. In C73, pp.479-482.

(I9) WHELER R.E. Random variable generators. In APL Quote Quad, vol.4, n°3, April 1973, pp.7-16.

(I10) WIEDMAN C. Step towards an APL compiler. In C79, pp.321-328.

(I11) HUBERT J.J., LAZOTTE D.C. The generalized inverse. In APL Quote Quad, vol.7, n°2, Sum. 1976, pp.19-25.

(I12) HASSIT A., LYON L.E. Efficient evaluation of array subscripts of arrays. IBM J. Res. & Develop., Jan. 1972.

(I13) JENKINS M.A. Translating APL, an empirical study. In C75, pp.192-200.

(I14) ASHCROFT E.A. Towards an APL compiler. In C74, pp.28-38.

(I15) MILLER T.C. Tentative compilation - A design for an APL compiler. In C79, pp.88-95.

(I16) GUIBAS L.J., WYATT D.K. Compilation and delayed execution in APL. Fifth POPL, ACM, 1978, pp.1-8.

(I17) VAN BREE K.A., MUNSEY G.J., JOHNSTON R.L., VAN DYKE E.J., LEONG W.W. Hewlett Packard Journal, July 1977.

(I18) TUSERA D. Description d'une méthode d'interprétation du langage APL par les attributs sémantiques. Thèse de Docteur-Ingénieur, Paris VI, Juin 1974.

6 - EXTENSIONS DIVERSES

a - **Graphique :**

(D1) BARON S., BARTELS S.R., MARTIN G. Programmes graphiques pour terminal Tektronix 4013. Saclay, CISI, Mars 1974.

(D2) OUANES M. APLIXI, opérateurs graphiques, manuel de référence. APLIXI, (Paris, 1978).

(D3) GALBRAITH D.S. Primitive functions for graphics in APL. In APL Quote Quad, vol.7, n°1, Summer 1976, pp.27-36.

b - Fichiers :

(F1) APLPLUS Workspace documentation. User's guide. Bethesda, Maryland, Scientific Time Sharing Corporation, 1973.

(F2) JURAN, MOORE, ORNDORFF, RICE PCS shared file system. In APL Quote Quad, vol.5, n°1/3, Spring 1974, pp.5-16.

(F3) MARTIN H. Etude et réalisation d'un système de gestion de fichiers APL. Thèse de Docteur-Ingénieur, Toulouse, Université P. Sabatier, Mai 1975.

c - Variable partagée :

(V1) IBM TSIO program reference manual. IBM, ref. SH20-1463, 1973.

d - Tableaux généralisés :

(T1) BROWN J.A. Evaluating extensions to APL. In C79, pp.148-155.

(T2) GHANDOUR Z., MEZEI J. General arrays, operators and functions. IBM j. of res. & develop., July 1973.

(T3) GULL W.E., JENKINS M.A. Recursive data structures in APL. In publ. of the ACM, vol.22, n°1, Jan. 1979, pp.79-96.

(T4) ALFONSECA M., TAVERA M.L. Extension of APL to tree-structured information. In C76, pp.1-23.

(T5) EDWARDS E.M. Generalized arrays (lists) in APL. In C73, pp.99-105.

(T6) MURRAY R.C. On tree structure extensions to the APL language. In C73, pp.333-338.

(T7) VASSEUR J.P. Extension of APL operators to tree-like data structures. In C73, pp.457-464.

(T8) BERNECKY B., IVERSON K.E. Operators and enclosed arrays. In C80, pp.319-331.

(T9) MORE T. The nested rectangular array as a model of data. In C79, pp.55-73.

(T10) PIERRE M., PIERRE P. A relationnal data base using generalized arrays and data base primitives. In C79, pp.102-109.

(T11) GULL W.E., JENKINS M.A. Recursive data structures and related control mechanisms. In C76, pp.201-210.

(T12) SMITH R.A. Nested arrays. APL*PLUS, S.T.S.C., 1981.

e - Sécurité et protection des applications :

(P1) LINK D.A., GARDNER M.W. Deferred execution : an "ACE" of an application. In C79, pp.1-7.

(P2) RYAN J.L. Secure application within an APL environment. In C79, pp.407-414.

(P3) PUCKET T.H., Improved security in APL application packages. In C74, pp.438-441.

(P4) MAYBURRY W.J. Programming aids in XEROX-APL. In C74, pp.307-315.

(P5) KLINE E.M. Technical notes on variable control. In APL Quote Quad, vol.4, n°4, June 1973, p.21.

(P6) SLIGOS APL.S, manuel de l'utilisateur. Sligos, Puteaux.

(P7) GROSSMAN R. Programmed error recovery for APLSV. In APL Quote Quad, vol.7, n°3, Fall 1976, pp.7-11.

(P8) BURRIL J.H. Some notes on handling errors in APL. In APL Quote Quad, vol.9, n°3, March 1979, pp.44-47.

(P9) GILMORE J.C., PUCKET T.H. A latent expression exception handling system. In C79, pp.244-248.

(P10) AIGRAIN Ph. A propos du contrôle d'erreur en APL. Journées d'étude APL : Les systèmes d'information, AFCET, Mai 1979.

7 - GESTION DE MEMOIRE

(M1) EHRMANN R. L'allocation dynamique de la mémoire des ordinateurs. Paris, Dunod, 1972.

(M2) GELENBE, LENFANT, BRANDWAJN, POTIER Analyse d'un algorithme de gestion simultanée, mémoire centrale, disque de pagination. IRIA, rapport de recherche n°19, 1973.

(M3) ARMINGAUD F.D. Gestion de données en APL. Ecole d'été de l'AFCET, Neuchâtel, 1972.

(M4) MACON H.P. Virtual APL. In APL Quote Quad, vol.5, n°1/3, Spring 1974, pp.17-23.

(M5) DENNING P.J. Virtual memory. In Computing Surveys, vol.2, n°3, Sept. 1970, pp.153-189.

8 - BIBLIOGRAPHIE GENERALE

(G1) SEEDS G.M. Fuzzy floor and ceiling. In APL Quote Quad, vol.3, n°4, Winter 1974, pp.66-68.

(G2) AHO A.V., ULLMAN J.D. Principles of compiler design. Addison-Wesley Publishing Company, April 1979.

(G3) MAHL R. Algorithmique et structures de données. Cours d'Informatique Software. Saint-Etienne, Ecole des Mines, (1974).

(G4) PAIR C. Structures de données et algorithmes fondamentaux. Nancy, Ecole des Mines, 1974.

(G5) BERTIN C. Conception d'un superviseur de temps partagé destiné à l'implémentation d'un interprète APL sur un miniordinateur. Rapport de DEA, Syst. et Res. Informatiques. Saint-Etienne, Ecole des Mines, Oct. 1977.

(G6) WEGNER P. Programming with ADA : an introduction by means of graduated examples. P.-H., Englewood Cliffs, New Jersey, 1980.

(G7) UNITED STATES DEPARTMENT OF DEFENSE Reference manual of the ADA programming language. July 1980.

(G8) NAKACHE M. Réalisation d'un noyau de système de gestion de base de données relationnelle sous APL. Thèse de Docteur-Ingénieur. Saint-Etienne, Ecole des Mines, Juin 1976.

(G9) BERTIER M. Gestion des fonctions utilisateur d'un interprète APL. Rapport de DEA, Syst. et Res. Informatiques, Saint-Etienne, Ecole des Mines, 1979.

(G10) GUIBOUD-RIBAUD S. Architecture des systèmes : une vue synthétique logicielle et matérielle. Saint-Etienne, Ecole des Mines, Juil. 1978.

(G11) KNUTH D.E. The art of programming. Vol.1 : Fundamental algorithms. Reading, Addison-Wesley, 1968.

9 - DOCUMENTATION TECHNIQUE APL/857

(A1) BERTIN C. APL/857 : La machine d'exécution. Element Design Specifications, Components, Tome III.2, EMSE, Déc. 1980.

(A2) BERTIN C. APL/857 : Trouble shooting guide. EDS, Tome III.5, EMSE, Janv. 1981.

(A3) MIREAUX F. APL/857 : Le superviseur monoconsole. EPS, EDS, Tomes I,II,III, EMSE, Janv. 1980.

(A4) GIRARDOT J.J. APL/857 : Les nouveaux aspects de la version 2. EMSE, Août 1980.

(A5) BERTIN C., GIRARDOT J.J. APL/857 : Les nouveaux aspects de la version 3. EMSE, Sept. 1980.

(A6) BERTIN C., GIRARDOT J.J. APL/857 : Architecture de l'interprète. EDS, Tome I, EMSE, Janv. 1979.

(A7) BERTIN C., MIREAUX F. APL/857 : Noyau et services de l'interprète. EDS, Tome III.0, Components, EMSE, Fév. 1980.

(A8) BERTIER M., BERTIN C., GIRARDOT J.J., MIREAUX F. L'éditeur de texte T.N.T. Spécifications externes et internes, EMSE, Mars 1979.

(A9) BERTIN C. APL/857 : Le préformateur disque DSKFMT. Spécifications internes et externes, EMSE, Sept. 1980.

(A10) BERTIER M., GIRARDOT J.J. L'éditeur de fonctions d'APL/857, manuel de référence. EMSE, Fév. 1980.

(A11) BERTIN C., GIRARDOT J.J. APL/857 : Fonctions et variables système, manuel de référence. EMSE, Fév. 1980.

(A12) BERTIN C. APL/857 : La page de réentrance, description des données de l'interprète. EDS, tome II.0, EMSE, Déc. 1980.

(A13) MIREAUX F. APL/857 : La machine de commandes. EDS, Tome III.1, EMSE, Nov. 1980.

(A14) MIREAUX F. APL/857 : L'éditeur de fonctions. EDS, Tome III.3, EMSE, Fév. 1981.

(A15) BERTIN C. APL/857 : Macrodéfinitions. EDS, Tome III.4, EMSE, Nov. 1980.

(A16) BERTIN C. APL Core Image Builder : l'éditeur de liens arborescent, chargeur du système APL/857. Spécifications internes et externes. EMSE, Fév. 1978.

(A17) BERTIN C. APL/857 et les tableaux de tableaux. Note introductive. EMSE, Juil. 1980.

(A18) MIREAUX F. Les fichiers d'APL/857. Spécifications externes. EMSE, Mars 1981.

(A19) BRAFFORT P., THILLIEZ F. Rapport final des tests de qualité faits à Fontenay-aux-roses. GFI Automatique, Philips Data System, Paris, Mars 1981.

(A20) BRAFFORT P., THILLIEZ F. Logiciel de contrôle et d'évaluation d'APL/857, manuel de l'utilisateur. GFI Automatique, Paris, Nov. 1980.

ANNEXES

A N N E X E 1

INTERFACE APL-FORTRAN

A) - Description de l'exemple :

L'exemple choisi est celui d'un sous-programme Fortran évaluant les coefficients du **polynôme caractéristique** $P(\lambda)$ d'une matrice carrée d'ordre n : A(n,n). Il fournit aussi le **déterminant** de A, et, selon sa régularité, soit la **comatrice transposée** de A, soit son **inverse** . La méthode employée est l'itération de SOURIAU-LIVERIER (méthode instable et peu efficace en $t=o(n^4)$).

Dans cet algorithme, à partir de A(n,n), on calcule (tr est la fonction trace, I la matrice identité) :

$A_1 = A$	$t_1 = -\,tr(A_1)$	$B_1 = A_1 + t_1 \times I$
$A_2 = B_1 \times A$	$t_2 = -\frac{1}{2}\,tr(A_2)$	$B_2 = A_2 + t_2 \times I$
...	...	...
$A_n = B_{n-1} \times A$	$t_n = -\frac{1}{n}\,tr(A_n)$	$B_n = A_n + t_n \times I$

Les scalaires t_i sont les coefficients du polynôme caractéristique normalisé :

(1) $P = (1, t_1, t_2, \ldots, t_{n-1}, t_n)$

D'après le théorème de Cayley-Hamilton :

(2) $P(A) = B_n = 0$

On obtient d'autre part :

(3) $\det(A) = -t_n$ et $(\text{com } A)^t = B_{n-1}$

Ce qui donne la condition de régularité : $t_n \neq 0$

(4) $A^{-1} = -\frac{1}{t_n}.B_{n-1}$

B) - Réalisation :

Elle comprend deux sous-programmes de service MULT(A,B,R,n) et TRACE(A,n) appelés par le sous-programme LAMBDA(A,P,INV,B,n).

1°) Construction de la fonction machine LAMBDA

Cette construction comprend six phases :

```
$:%LINK    I=LAMBDA,U=CH,SSP1=MULT,SSP2=TRACE,ED=NO
MES
MES        INTERFACE APL-FORTRAN EMSE C.B. 07/81
MES        1)  CONSTRUCTION ET ASSEMBLAGE DU LANCEUR
MES        2)  COMPILATION DU SOUS-PROGRAMME 'PRINCIPAL'
MES        3)  COMPILATION DES SS-PROGRAMMES DE SERVICE
MES        4)  EDITION DE LIENS STANDARD
MES        5)  RELECTURE DE L'IMAGE MEMOIRE EN FORMAT OBJET
MES        6)  SESSION A.C.I.B. DE CONSTRUCTION DE LA FONC. MACHINE
```

La première phase est l'édition et l'assemblage d'un sous-programme d'un mot qui contiendra l'adresse du point d'entrée du sous-progamme principal Fortran.

```
           IDENT     STARTR  CH.B 20/07/80
           ENTRY     F:FCT
           EXTRN     LAMBDA
F:FCT      DATA      LAMBDA
           END       F:FCT
```

Phase 1 : Construction et assemblage du lanceur.

Ensuite, on passe aux compilations Fortran du sous-programme principal et des sous-programmes ancillaires.

```
           REAL FUNCTION TRACE(A,N)
           REAL A(N,N)
           TRACE=0.
           DO 1 I=1,N
     1     TRACE=TRACE+A(I,I)
           RETURN
           END


           SUBROUTINE MULT(A,B,R,N)
           REAL A(N,N),B(N,N),R(N,N)
           REAL RR
           DO 1 I=1,N
           DO 1 J=1,N
           RR=0.
           DO 2 K=1,N
     2     RR=RR+A(I,K)*B(K,J)
     1     R(I,J)=RR
           RETURN
           END
```

Phase 3 : Compilation des sous-programmes ancillaires.

```
      IDENT     LAMBDA    CB/07.81
      SUBROUTINE LAMBDA(A,P,INV,B,N)
      INTEGER IT
      REAL    T,TN,TRACE,BIJ
      REAL    A(N,N),B(N,N),INV(N,N),P(N)
C     METHODE DE SOURIAU-LIVERIER:
C     ----------------------------
C     CALCULE LE POLYNOME CARACTERISTIQUE DE A DANS P
C     LE DERNIER COEFFICIENT EST LE DETERMINANT DE A
C     FOURNIT DANS INV LA COMATRICE TRANSPOSEE OU L'INVERS
      E
C     DE A SELON LA REGULARITE DE A
C     B(N,N) EST UNE MATRICE DE TRAVAIL INDISPENSABLE
C     D'APRES LE THEOREME DE CAYLEY-HAMILTON : B=P(A)=0
C     B=I & INV=A
      DO 10 I=1,N
      DO 10 J=1,N
      B(I,J)=0.
      INV(I,J)=A(I,J)
      IF(I.EQ.J) B(I,I)=1.
10    CONTINUE
C     ITERATION DE SOURIAU-LIVERIER :
C
      NN1=N-1
      DO 1 IT=1,NN1
      CALL MULT (B,A,INV,N)
      T=(-1./IT)*TRACE(INV,N)
      P(IT)=T
C     (B.I=A.I+T.I)
C
      DO 2 I=1,N
      DO 2 J=1,N
      BIJ=INV(I,J)
      IF(I.EQ.J) BIJ=BIJ+T
2     B(I,J)=BIJ
1     CONTINUE
C     ORDRE N-1
C
      CALL MULT(B,A,INV,N)
      TN=-(1./N)*TRACE(INV,N)
      T=TN
      P(N)=T
      IF(ABS(T).LT.3E-7) T=-1.
      DO 20 I=1,N
      DO 20 J=1,N
      BIJ=INV(I,J)
      INV(I,J)=-B(I,J)/T
      IF(I.EQ.J) BIJ=BIJ+TN
      B(I,J)=BIJ
20    CONTINUE
      RETURN
      END
```

Phase 2 : Compilation du sous-programme principal.

Une édition de liens standard crée une image mémoire comprenant les programmes de service demandés par Fortran (phase 4). Cette image mémoire est relue sous un format objet, où ne subsiste aucune référence d'externe. Enfin ce code objet est intégré avec le module spécial d'interface (APLFOR) dans la branche spéciale des fonctions machine du système APL/857 (Phase 6). Cette deuxième édition de liens est indépendante de la précédente et permet d'éviter les conflits de noms de points d'entrée.

2°) Chargement de la fonction machine dans la zone de travail :

Le système APL est activé : l'utilisateur charge sous le nom LAMBDA une fonction machine monadique :

```
APL\857 V-04.0  19/10/81  11.37.18.
WORKING STORAGE OF   564  K-BYTES

      )FXLOAD LAMBDA 1
      MAT←⍉3 3⍴2 3 0 1.5 ¯2 2 0 1 .5
      POL←.1 .1 .1
      INVERSE←3 3⍴.2
      TRAV←3 3⍴.1
      ⎕PP←4
      MAT
2    1.5   0
3    ¯2    1
0    2     .5

      LAMBDA(MAT;POL;INVERSE;TRAV;,3)
      INVERSE
.3636      9.091E¯2   ¯.1818
.1818      ¯.1212     .2424
¯.7273     .4848      1.03
      ∧/,INVERSE=⌹MAT
1
      TRAV
¯1.49E¯8   0          0
0          ¯1.49E¯8   0
0          0          ¯1.49E¯8
      )OFF
CONNECTED TIME 00.03.49
CPU TIME       00.00.05
```

Chargement et activation de la fonction machine.

3°) Interface simplifié :

Cet interface simplifié ne tient pas compte des problèmes de transposition, de type, de partage et de résidence mémoire etc.

```
                                        IDENT     APLFOR  17.07.81 CH-B.
                          *
                                        ENTRY     APLFOR
                          *
                                        EXTRN     RQST,ADC
                          *
                                        EXTRN     STARTR

                                        NLIST
                                        LIST
                                        NLIST
                                        LIST
              000A        NIVPD2    EQU       10
              5000        ADDPD2    EQU       /5000

                          *
                          **
                          ***       APPEL SSP FORTRAN
                          **
                          *
              0000 R      APLFOR    EQU       *
                          *-----
0000    0206                        LDK       A2,6
0002    0300                        LDK       A3,0
0004    8720    03EC X              LDK,L     A7,IP
0008    81A0    08CA X              LDK,L     A9,ALIEN2
000C    F6A1    0000 X              CF        A14,RQST
                          *
0010    A041    03EC X              CM        IP
0014    0500                        LDK       A5,0

              0016 R      BICLAR    EQU       *
0016    8140    03EC X              LD        A1,IP
001A    E940    08D2 X              CW        A1,ELCNT2
001E    5600 F                      RF(NL)    NOPAR

0020    F6C1    08CA X              CALL      ALIEN2
0024    B940    0BFA X              ML        2,EDGAR+4
0028    0700                        LDK       A7,0
002A    F6A1    0000 X              CF        A14,ADC
002E    8755    0748 X              ST        A7,I,A5
0032    1502                        ADK       A5,2
0034    9041    03EC X              IM        IP
0038    5F24                        RB        BICLAR

              003A R      NOPAR     EQU       *
003A    8420    0748 X              LDK,L     A4,I
003E    F6C1    0000 X              CALL      STARTR
0042    F6C1    0352 X              CALL      FXRTN

0046                                END
```

Listage de l'interface.

A N N E X E 2

1	INVALID CONSTANT	33	DIRECTORY FULL
2	SYMBOL TABLE FULL	34	DISK I/O ERROR /HHHH
3	LINE TOO LONG	35	I/O ERROR /HHHH
4	DOMAIN ERROR	36	NOT WITH OPEN DEFINITION
5	SYNTAX ERROR	37	WS LOCKED
6	INDEX ERROR	38	FILE IS NOT AN APL WS
7	ID ERROR	39	STATE INTERRUPT
8	DEFN ERROR	40	BAD EVL NUMBER
9	VALUE ERROR	41	SI DAMAGED
10	WS FULL	42	DOMAIN LIMIT
11	LABEL ERROR	43	RANK LIMIT
12	DEPTH ERROR	44	LENGTH LIMIT
13	LENGTH ERROR	45	SIZE LIMIT
14	RANK ERROR	46	UNABLE TO RESUME EXECUTION
15	NONCE ERROR	47	CHARACTER ERROR
16	COMMAND UNKNOWN	48	INTERRUPT
17	INCORRECT COMMAND	49	INVALID LOGICAL UNIT
18	#IO IMPLICIT ERROR	50	INCORRECT FILE-CODE
19	#CT IMPLICIT ERROR	51	WS SEALED
20	#PP IMPLICIT ERROR	52	NAME NOT FOUND
21	#PW IMPLICIT ERROR	53	NO SPACE
22	#RL IMPLICIT ERROR	54	FILE ACCESS ERROR
23	- UNUSED -	55	FILE FULL
24	#LX IMPLICIT ERROR	56	FILE TIE ERROR
25	- UNUSED -	57	FILE NAME ERROR
26	- UNUSED -	58	FILE INDEX ERROR
27	- UNUSED -	59	FILE EMPTY
28	#WI IMPLICIT ERROR	60	FILE VALUE ERROR
29	USER NOT IN SYSTEM	61	FILE RESERVATION ERROR
30	WS NOT FOUND	62	FILE TIED
31	INCORRECT DISK FILE CODE	63	FILE TIE QUOTA USED UP
32	DISK FULL	64	FILE SYSTEM ERROR

Liste des messages d'erreur du système APL/857.

A N N E X E 3

TABLEAUX GENERALISES

Nous présentons dans ce troisième annexe une session commentée d'utilisation des primitives et opérateurs proposés par Gull et Jenkins en 1979 (T3) :

A) - Liste des primitives et opérateurs :

PRIMITIVES MONADIQUES :

>	*DECOUVRIR*
⍋	*DESCENDRE*
⍒	*DETERMINER*
⍕	*EDITER*
<	*ENFERMER*
⍪	*JOINDRE*
⍋	*MONTER*
,	*PREORDONNER*

PRIMITIVES DYADIQUES :

∈	*APPARTENIR*
⍷	*ATTEINDRE*
⍪	*CHAINER*
⍸	*CHOISIR*
=	*COMPARER*
,	*COUPLER*
⍋	*MONTER*
⍪	*PARTITIONNER*
⍳	*RECHERCHER*
⍋	*TRANCHER*
⍤	*VISER*

OPERATEURS :

⍤	*APPLIQUER*
⍣	*APPLIQUER RECURSIVEMENT*

B) - Editions des tableaux généralisés :

La première fonction d'édition que nous proposons est une généralisation de la primitive monadique format appliquée aux objets structurés (édition dite "tacite" de la fig. 12) :

```
    ∇  R←SEDI V;⎕IO;T;M;RK;RV;IMAX;JMAX;L;I;J;RR;D
[1]   ⍝ EDITION DE STRUCTURES.      FONCTION    R←⍕V
[2]   ⍝ D EST LE DECALAGE ENTRE STRUCTURES.
[3]    ⎕IO←0 ◇ ⎕TRAP←¯4 ◇ D←3
[4]   ⍝ EST-CE UNE STRUCTURE VIDE ?
[5]    →(0≠⍴,V)/NONVIDE
[6]   ⍝ OUI, LE RESULTAT EST UNE MATRICE VIDE.
[7]    R←0 0⍴'' ◇ →0
[8]   ⍝ OBJET NON VIDE. EST-CE UN OBJET NON STRUCTURE ?
[9]   NONVIDE: ⎕TRAP←0 ◇ →(6=⎕TY V)/STRU
[10]  ⍝ TABLEAU SIMPLE. ON UTILISE ⍕, ON GENERE UNE MATRICE
[11]   RR←1 1,⍴R←⍕V ◇ RR←(×/¯1↓RR),¯1↑RR ◇ R←RR⍴R ◇ →0
[12]  ⍝ STRUCTURE. ON CONSIDERE 4 CAS: SCALAIRE, VECT, MATRICE,
       AUTRE
[13]  STRU: →(0 1 2=⍴RV←⍴V)/SCALAIRE,VECTEUR,MATRICE
[14]  ⍝ RANG > 2. ON SE RAMENE A UN OBJET DE RANG 2.
[15]   V←(RV←(×/¯1↓RV),¯1↑RV)⍴V ◇ →MATRICE
[16]  ⍝ CAS D'UN SCALAIRE.
[17]  SCALAIRE: R←' ',(SEDI>V),' ' ◇ →0
[18]  ⍝
[19]  ⍝   CAS D'UN VECTEUR. ON EFFECTUE DEUX PASSES:
[20]  ⍝       - EDITION DE CHAQUE ELEMENT EN CONSERVANT LA TAILLE
[21]  ⍝       - GENERATION D'UNE MATRICE CARACTERE COMME RESULTAT.
[22]  VECTEUR: T←0⍴<RK←0 2⍴0
[23]  L1: →(0=⍴V)/N2 ◇ RK←RK⍪⍴M←SEDI>V[0] ◇ V←1↓V ◇ T←T,<M ◇ →L1
[24]  ⍝ CALCULER LE NOMBRE DE LIGNES DU RESULTAT.
[25]  N2: RK←(⌈/RK[;0]),-D+RK[;1] ◇ R←RK[0;]↑>T[0] ◇ RK[;1]←¯1+
       RK[;1]
[26]  ⍝ SECONDE BOUCLE POUR CREATION EFFECTIVE DU RESULTAT.
[27]  LP2: →(0=⍴T←1↓T)/ELQ1 ◇ RK←1 0↓RK ◇ R←R,RK[0;]↑>T[0] ◇ →LP
       2
[28]  ⍝
[29]  ⍝   EDITION D'UNE MATRICE
[30]  MATRICE: V←,V ◇ T←0⍴<RK←0 2⍴0
[31]  ⍝ PREMIERE BOUCLE PUR EDITER CHAQUE ELEMENT.
[32]  M1: →(0=⍴V)/M2 ◇ RK←RK⍪⍴M←SEDI>V[0] ◇ V←1↓V ◇ T←T,<M ◇ →M1
[33]  ⍝ RECREATION DE LA STRUCTURE ORIGINELLE.
[34]  ⍝ CALCULER LA HAUTEUR DE CHAQUE LIGNE.
[35]  M2: T←RV⍴T ◇ RK←(RV,2)⍴RK ◇ RK[;;0]←⍉(⍋RV)⍴⌈/RK[;;0]
[36]  ⍝ CALCULER LA LARGEUR DE CHAQUE COLONNE.
[37]   RK[;;1]←-RV⍴⌈⌿D+RK[;;1] ◇ IMAX←RV[I←0] ◇ JMAX←RV[1]
[38]  ⍝ GENERERLE RESULTAT PAR DEUX BOUCLES IMBRIQUEES.
[39]   R←(0,(JMAX-1)+-+/RK[0;;1])⍴' '
[40]  LQ1: L←((⌈/RK[I;;0]),J←0)⍴''
[41]  LQ2: L←L,RK[I;J;]↑>T[I;J] ◇ →(JMAX≤J←J+1)/ELQ2 ◇ L←L,' ' ◇
       →LQ2
[42]  ELQ2: R←R⍪L ◇ →(IMAX≤I←I+1)/ELQ1 ◇ R←R⍪' ' ◇ →LQ1
[43]  ELQ1: R←' ',R,' '
    ∇
```

Figure 12 : Listage de la primitive d'édition tacite

Ainsi, le résultat de l'expression suivante sera édité sous la forme :

```
      M2←('RESULT =';2 5ρ(1 1;2 2ρ'ABCD';24;'Q';4 2ρι8;7))
      SEDI M2
RESULT =          1 1      AB      24      Q        1  2
                           CD                       3  4
                                                    5  6
                                                    7  8

                    7      1 1      AB      24          Q
                                    CD
```

Nous utiliserons en outre une seconde fonction d'édition, plus précise quant à l'organisation structurelle de l'objet édité (édition dite "loquace" de la fig. 13) :

```
      P M2
$ STRUCTURE $     ρ ↔ 2
[1]
|  RESULT =
[2]
|  $ STRUCTURE $     ρ ↔ 2  5
|  [1;1]
|  |  1 1
|  [1;2]
|  |  AB
|  |  CD
|  [1;3]
|  |  24
|  [1;4]
|  |  Q
|  [1;5]
|  |     1  2
|  |     3  4
|  |     5  6
|  |     7  8
|  [2;1]
|  |  7
|  [2;2]
|  |  1 1
|  [2;3]
|  |  AB
|  |  CD
|  [2;4]
|  |  24
|  [2;5]
|  |  Q
|  $ END $
$ END $
```

```
    ∇  P X;⎕IO
[1]   ⍝ EDITION D'UN TABLEAU GENERALISE ARBORESCENT
[2]    ⎕IO←0
[3]    0 SH1 X
    ∇
```

```
    ∇  D SH1 V;W;X;Y ⍝ IMPRESSION
[1]    ⎕TRAP←¯4
[2]    →(6=⎕TY V)/STR ◇ →ARR
[3]    →VALERR
[4]   ⍝ CAS TABLEAU
[5]   ARR: (⍴V)SH2,V←⍕V
[6]    →0
[7]   STR: X←(D⍴'|   '),'$ STRUCTURE $'
[8]    →(0=⍴⍴V)/SSCAL
[9]    W←⍴V ◇ V←,V ◇ Y←0
[10]   X,'     ⍴ ←→ ',⍕W
[11]  LP: →(Y≥⍴V)/ND
[12]   (D⍴'|   '),(X≠' ')/X←'[',(1↓,';',⍕((⍴W),1)⍴1+W⊤Y),']'
[13]   (D+3)SH1>V[Y] ◇ Y←Y+1 ◇ →LP
[14]   ((D+3)⍴'|   '),'$ NO VALUE $' ◇ Y←Y+1 ◇ →LP
[15]  SSCAL: X
[16]   (D+3)SH1>V ◇ →ND
[17]   ((D+3)⍴'|   '),'$ NO VALUE $'
[18]  ND: (D⍴'|   '),'$ END $'
[19]   →0
[20]  VALERR: (D⍴'|   '),'$ NO VALUE $'
    ∇
```

```
    ∇  R SH2 V;U;W;X
[1]    →(1<⍴R)/KTAB
[2]    (D⍴'|   '),V
[3]    →0
[4]   KTAB:
[5]    U←1↑R ◇ W←1↓R
[6]   LP: →(U=0)/ND
[7]    W SH2(×/W)⍴V
[8]    →(U=1)/0
[9]    V←(×/W)↓V ◇ U←U-1 ◇ X←¯2+⍴R
[10]  LQ: →(X≤0)/LP
[11]   D⍴'|   ' ◇ X←X-1 ◇ →LQ
    ∇
```

Figure 13 : Listage de la primitive d'édition loquace

C) - Session commentée :

Le résultat de chaque expression est édité sous les deux formes :

Forme tacite **Forme loquace**

Création d'un vecteur numérique :

```
      □←A←11 12
11  12
```

d'un scalaire généralisé :

```
      □←B←<A
 11  12
```

```
      □←B←<A
$ STRUCTURE $
|  11  12
$ END $
```

d'une matrice vide :

```
      □←C←2 0ρ0
```

d'un pseudo-scalaire généralisé de rang 3 :

```
      □←D←1 1 1ρ<C
```

```
      □←D←1 1 1ρ<C
$ STRUCTURE $    ρ ↔ 1  1  1
[1;1;1]
$ END $
```

d'une matrice caractère :

```
      □←E←2 2ρ'ABCD'
AB
CD
```

L'opérateur ⍪ fait monter un niveau "structure" sur chaque élément de E :

```
      □←F←⍪E
A   B

C   D
```

```
      □←F←⍪E
$ STRUCTURE $    ρ ↔ 2  2
[1;1]
|  A
[1;2]
|  B
[2;1]
|  C
[2;2]
|  D
$ END $
```

Concaténation de deux structures :

```
      □←G←(<⍳3),<E
1  2  3      AB
             CD
```

```
      □←G←(<⍳3),<E
$ STRUCTURE $     ρ ↔ 2
[1]
|  1  2  3
[2]
|  AB
|  CD
$ END $
```

Application de la primitive monadique ρ aux éléments de G :

```
      □←H←¨ρG
3     2  2
```

```
      □←H←¨ρG
$ STRUCTURE $     ρ ↔ 2
[1]
|  3
[2]
|  2  2
$ END $
```

De même, avec la primitive monadique ⍳ :

```
      □←I←¨⍳⍳3
1     1  2     1  2  3
```

```
      □←I←¨⍳⍳3
$ STRUCTURE $     ρ ↔ 3
[1]
|  1
[2]
|  1  2
[3]
|  1  2  3
$ END $
```

Application de la primitive dyadique ρ entre éléments de I et de B (il y a extension généralisée de B) :

```
      □←J←I¨ρB
11       11  12      11  12  11
                     12  11  12
```

```
      □←J←I¨ρB
$ STRUCTURE $     ρ ↔ 3
[1]
|  11
[2]
|     11  12
[3]
|     11  12  11
|     12  11  12
$ END $
```

Construction d'un vecteur structure à l'aide d'un masque de partitionnement :

```
      □←K←'ABCDEF'
ABCDEF
```

```
      □←L←0 1 0 0 1 0⌿K
BCD      EF
```

```
      □←L←0 1 0 0 1 0⌿K
$ STRUCTURE $     ρ ↔ 2
[1]
|  BCD
[2]
|  EF
$ END $
```

La primitive dyadique ↑ permet de transformer une matrice en le vecteur de ses lignes :

```
      □←M←3 4⍴⍳12
 1  2  3  4
 5  6  7  8
 9 10 11 12
```

```
      □←N←1↑M
 1  2  3  4  5  6  7  8  9 10 11 12
```

```
      □←N←1↑M
$ STRUCTURE $     ⍴ ↔ 3
[1]
|  1  2  3  4
[2]
|  5  6  7  8
[3]
|  9 10 11 12
$ END $
```

La fonction "descendre" (↓) réalise l'opération réciproque de la fonction "monter" (↑) :

```
      □←O←⍤↓(<1 0 0 1)⍀N
 1  2  3  4

 5  6  7  8

 9 10 11 12
```

```
      □←O←⍤↓(<1 0 0 1)⍀N
$ STRUCTURE $     ⍴ ↔ 3 2
[1;1]
|  1  2  3
[1;2]
|  4
[2;1]
|  5  6  7
[2;2]
|  8
[3;1]
|  9 10 11
[3;2]
|  12
$ END $
```

Application de ↑ aux vecteurs colonnes de l'objet O :

```
      □←P←⍤↓1↑⌽O
 1  2  3  4
 5  6  7  8
 9 10 11 12
```

```
      □←P←⍤↓1↑⌽O
$ STRUCTURE $     ⍴ ↔ 2
[1]
|  1  2  3
|  5  6  7
|  9 10 11
[2]
|  4
|  8
| 12
$ END $
```

```
      □←Q←1↑⌽M
 1  5  9  2  6 10  3  7 11  4  8 12
```

```
      □←Q←1↑⌽M
$ STRUCTURE $     ⍴ ↔ 4
[1]
|  1  5  9
[2]
|  2  6 10
[3]
|  3  7 11
[4]
|  4  8 12
$ END $
```

```
      □←R←⌽⍉(<1 1 0)⍤⍉Q
   1        2        3        4
5  9    6  10    7  11    8  12
```

```
      □←R←⌽⍉(<1 1 0)⍤⍉Q
$ STRUCTURE $     ρ ↔ 2  4
[1;1]
|  1
[1;2]
|  2
[1;3]
|  3
[1;4]
|  4
[2;1]
|  5  9
[2;2]
|  6  10
[2;3]
|  7  11
[2;4]
|  8  12
$ END $
```

```
      □←S←⍤⌽⍤⍉1↑R
1  2  3  4        5   6   7   8
                   9  10  11  12
```

```
      □←S←⍤⌽⍤⍉1↑R
$ STRUCTURE $     ρ ↔ 2
[1]
|     1  2  3  4
[2]
|     5   6   7   8
|     9  10  11  12
$ END $
```

Les primitives "coupler" et "chaîner" :

```
      □←T←3 2ρ((⍳0)⍪E⍪A⍪11 12 13⍪B⍪D)
                    AB
                    CD

 11  12    11  12  13

11  12
```

```
      □←T←3 2ρ((⍳0)⍪E⍪A⍪11 12 13⍪B⍪D)
$ STRUCTURE $     ρ ↔ 3  2
[1;1]
|
[1;2]
|  AB
|  CD
[2;1]
|  11  12
[2;2]
|  11  12  13
[3;1]
|  $ STRUCTURE $
|  |  11  12
|  $ END $
[3;2]
|  $ STRUCTURE $     ρ ↔ 1  1  1
|  [1;1;1]
|  $ END $
$ END $
```

```
      □←U←E⌽T¯,⍳0
AB
CD

          11  12    11  12  13

        11  12
```

```
      □←U←E⌽T¯,⍳0
AB    $ STRUCTURE $      ρ ↔ 3
CD    [1]
      |  AB
      |  CD
      [2]
      |  $ STRUCTURE $      ρ ↔ 3  2
      |  [1;1]
      |  |
      |  [1;2]
      |  |  AB
      |  |  CD
      |  [2;1]
      |  |  11  12
      |  [2;2]
      |  |  11  12  13
      |  [3;1]
      |  |  $ STRUCTURE $
      |  |  |  11  12
      |  |  $ END $
      |  [3;2]
      |  |  $ STRUCTURE $      ρ ↔ 1  1  1
      |  |  [1;1;1]
      |  |  $ END $
      |  $ END $
      [3]
      |
      $ END $
```

La primitive ⍪ crée le vecteur des feuilles de l'objet U, quand celui-ci est parcouru en préordre :

```
      □←V←⍪U
AB          AB      11  12      11  12  13      11  12
CD          CD
```

```
      □←V←⍪U
$ STRUCTURE $      ρ ↔ 8
[1]
|  AB
|  CD
[2]
|
[3]
|  AB
|  CD
[4]
|  11  12
[5]
|  11  12  13
[6]
|  11  12
[7]
[8]
|
$ END $
```

Application de la primitive ρ aux éléments du premier niveau de l'objet U :

```
      □←W←¨ρU
  2   2       3   2       0
```

```
      □←W←¨ρU
$ STRUCTURE $      ρ ↔ 3
[1]
|   2   2
[2]
|   3   2
[3]
|   0
$ END $
```

Application récursive de la primitive ρ sur toute la descendance de l'objet U :

```
      □←X←⍤¨ρU
2   2           0     2   2         0
                2         3
              2
```

```
      □←X←⍤¨ρU
$ STRUCTURE $      ρ ↔ 3
[1]
|   2   2
[2]
|   $ STRUCTURE $      ρ ↔ 3   2
|   [1;1]
|   |   0
|   [1;2]
|   |   2   2
|   [2;1]
|   |   2
|   [2;2]
|   |   3
|   [3;1]
|   |   $ STRUCTURE $
|   |   |   2
|   |   $ END $
|   [3;2]
|   |   $ STRUCTURE $      ρ ↔ 1   1   1
|   |   [1;1;1]
|   |   |   2   0
|   |   $ END $
|   $ END $
[3]
|   0
$ END $
```

La primitive "atteindre" (⍸) permet de sélectionner des sous-arbres en donnant leur chemin d'accès :

```
      U⍸<1
AB
CD
```

```
      □←Y←,⍤⍳⍴U
2  2     0     2  2     2     3     2     2  0     0
```

```
      □←Y←,⍤⍳⍴U
$ STRUCTURE $     ⍴ ↔ 8
[1]
|   2   2
[2]
|   0
[3]
|   2   2
[4]
|   2
[5]
|   3
[6]
|   2
[7]
|   2   0
[8]
|   0
$ END $
```

Sélection du second élément de l'objet U, et, dans le résultat obtenu, de l'élément d'indices (1 2) :

```
      U⌷2,⍤1 2
AB
CD
```

```
      U⌷2,⍤2 1
11  12
```

Un indice vide permet alors d'indexer un scalaire!

```
      U⌷(2;3 1;⍳0)
11  12
```

Sélection par la fonction "trancher" (⍀) : une écriture plus élégante de l'indexation :

```
      (2 3,⍤1)⍀T
  11  12      11  12
```

```
      (2 3,⍤1)⍀T
$ STRUCTURE $     ⍴ ↔ 2
[1]
|   11  12
[2]
|   $ STRUCTURE $
|   |   11  12
|   $ END $
$ END $
```

```
      (↑1 2)⍀T
AB
CD
```

```
      (↑1 2)⍀T
$ STRUCTURE $
|   AB
|   CD
$ END $
```

Appartenances généralisées :

```
      V∈<⍳0
0 1 0 0 0 0 0 1

      T∈U
 1 1
 0 0
 0 0

      (<T)∈U
1

      V∈<E
1 0 1 0 0 0 0 0
```

Recherche d'indices généralisée :

```
      V⍳T
 2  1
 4  5
 9  9
```

dernière page de la thèse

AUTORISATION DE SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974,

VU les rapports de présentation de M. J.J. GIRARDOT et M. NAKACHE

M. BERTIN Christian

est autorisé à présenter une thèse en soutenance pour l'obtention du diplôme de DOCTEUR-INGENIEUR, spécialité Systèmes et Réseaux informatiques

Fait à Saint-Etienne, le 24 novembre 1981

Le Président de l'INPG,

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

Le Directeur de l'EMSE,

M. HERMET
DIRECTEUR
ECOLE NATIONALE SUPERIEURE DES
MINES DE SAINT-ETIENNE